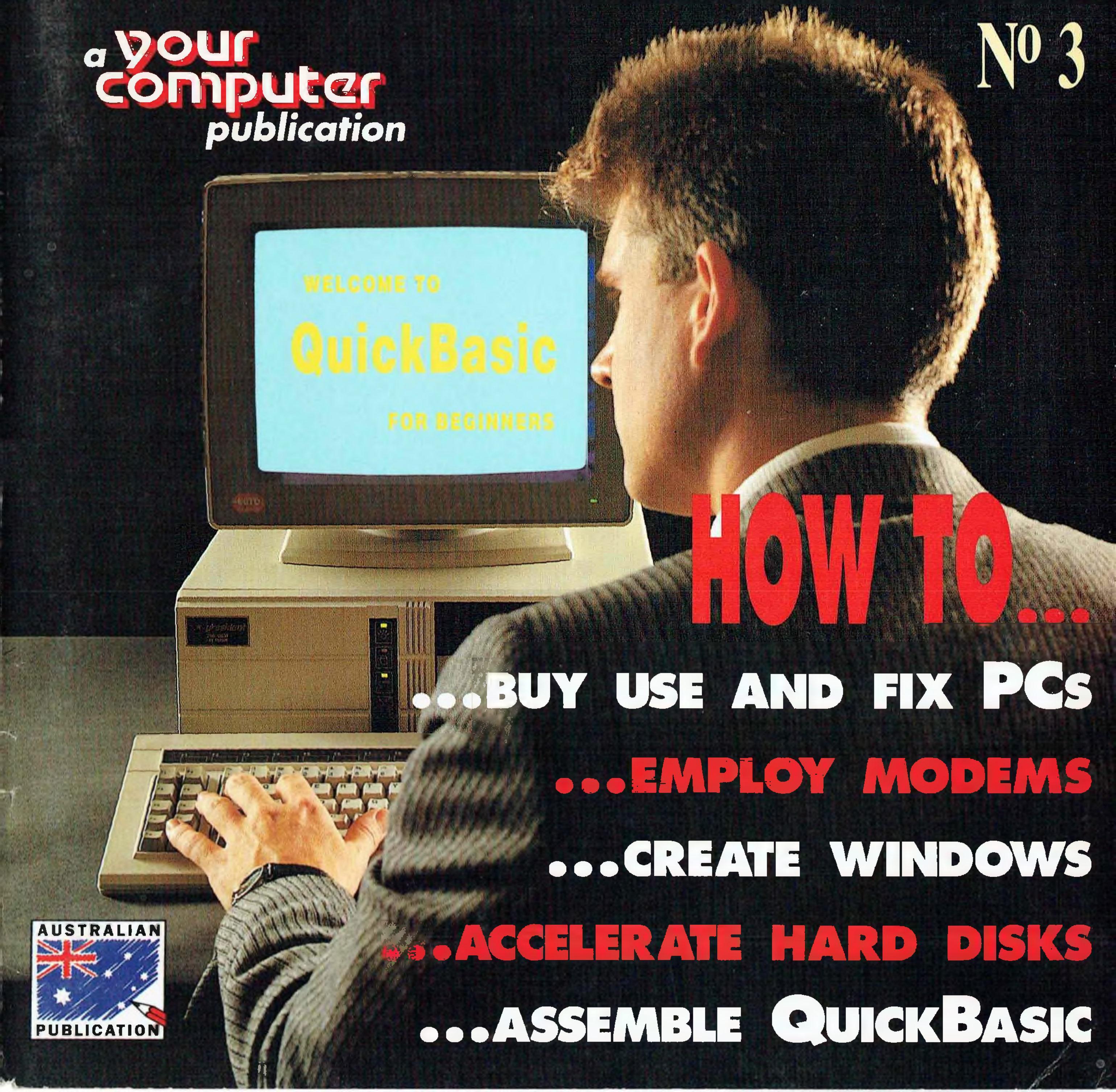
NDD

LEARNING WITH YOUR COMPUTER

$4.95
NZ $7.25
inc. GST

a your computer publication

Nº 3



HOW TO...

...BUY USE AND FIX PCs

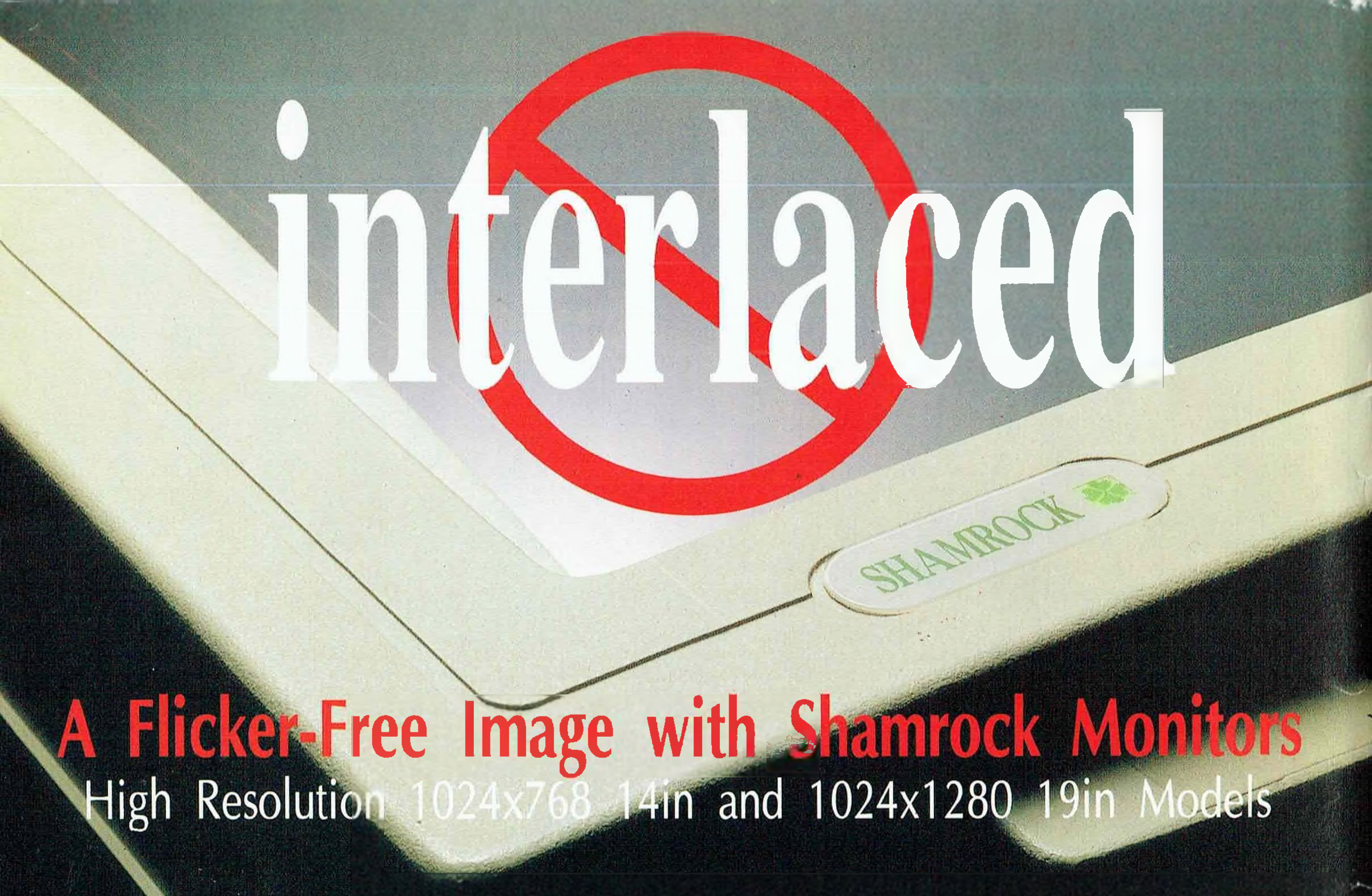
...EMPLOY MODEMS

...CREATE WINDOWS

...ACCELERATE HARD DISKS

...ASSEMBLE QUICKBASIC

AUSTRALIAN PUBLICATION



A Flicker-Free Image with Shamrock Monitors

High Resolution 1024x768 14in and 1024x1280 19in Models

SRC1401/1491* 14in HYPER VGA I

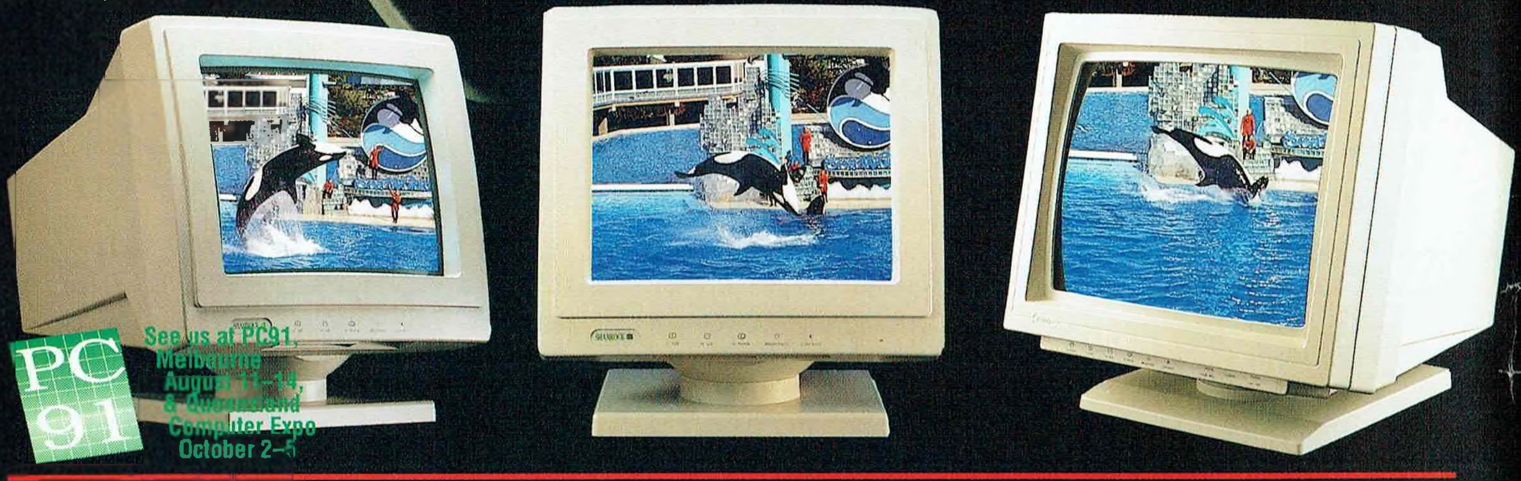
- ■ Horizontal Frequency: 28kHz - 40kHz
- ■ Resolution: 1024 x 768 Interlaced
- ■ Compatible with the full range of Super VGA modes plus Apple Mac II
- ■ 0.28 (*SCR1491 0.39) Dot Pitch CRT

SRC 1402 14in HYPER VGA II

- ■ Horizontal Frequency: 28kHz – 50kHz
- ■ Resolution: 1204 x 768 non-interlaced
- ■ Compatible with the full range of Super VGA modes plus Apple Mac II
- ■ 0.28 Dot Pitch CRT

SRC-1901 19in CAD/CAM MULTISCAN

- ■ Horizontal Frequency: 27kHz – 72kHz
- ■ Resolution: 1280 x 1024
- ■ DBF (Dynamic Beam Forming)
- ■ 0.31 Dot Pitch CRT
- ■ BNC & D-SUB (15 pin) input



PC 91

See us at PC91, Melbourne August 11–14, & Queensland Computer Expo October 2–5

OPTIMA COMPUTER TECHNOLOGY

OCT

HEAD OFFICE:
Unit 18, Metro Centre,
38-46 South St,
Rydalmere NSW 2116
Tel (02) 638 1222 Fax (02) 638 6324

QUEENSLAND:
Unit 8, Hudson Connection,
10 Hudson Road,
Albion Qld 4010
Tel (07) 262 8377 Fax (07) 262 8466

VICTORIA:
Zitech Electronics,
57 St. Hellier Street,
Heidelberg Heights Vic 3081
Tel (03) 459 7222 Fax (03) 458 3237

Your Computer monthly magazine is staffed by:

EDITOR
Jake Kennedy
TECHNICAL EDITOR
Mark Cheeseman
PRODUCTION EDITOR
Nina Stevens
ART DIRECTOR
Sally Anne Silveira
MANAGING EDITOR
Barry Tranter
PUBLISHER
Michael Hannan
EDITORIAL AND OFFICE SERVICES
Vera Popovska, 180 Bourke Road, Alexandria, NSW 2015. Tel: (02) 693 4173; Fax: (02) 693 9720.
ADVERTISING:
National Manager: San Sri, 180 Bourke Road, Alexandria, NSW 2015. Tel: (02) 693 9724; Fax: (02) 693 9720. **NSW:** (02) 693 6666. **Vic:** (03) 795 3666; Fax: (03) 701 1534. **Qld:** (07) 854 1119; Fax: (07) 252 3692. **SA:** (08) 352 6033; Fax: (08) 352 6033. **WA:** Tony Allen, Allen & Assoc, 1st Floor, CTA Building, 54 Havelock Street, West Perth 6005. Tel: (09) 321 2998; Fax: (09) 321 2940. **New Zealand:** Federal Publishing, 67-73 View Road, Glenfield, Auckland. Tel. (09) 443 0954; Fax: (09) 443 1326. **United Kingdom:** Peter Holloway, John Fairfax & Sons (Australia) Ltd., Associated Press House, 12 Norwich Street, London EC4A 1BH. Tel: (01) 353 9321; Fax: (01) 583 0348. **Hong Kong:** Adrian Batten, Headway Media Services Ltd, Room 2101, Causeway Bay Centre, 15-23 Sugar Street, Hong Kong. Tel: (852) 576 8002; Fax: (852) 890 4811.
PRODUCTION MANAGER
Brett Baker
PRODUCTION CO-ORDINATOR
Patrice Wohlnick
CIRCULATION MANAGER
Michael Prior

Distributed by Newsagents Direct Distribution Pty. Ltd., 150 Bourke Road, Alexandria, NSW 2015. Tel: (02) 693 9517. Distributed in New Zealand by Network Distributors Ltd., 67-73 View Road, Glenfield, Auckland. Tel: (09) 443 0245; Fax: (09) 443 0249.

Learning with Your Computer and *Your Computer* magazine are published by Federal Publishing Company, a partnership of General Newspapers Pty. Ltd., A.C.N. 000 117 322; Double Bay Newspapers Pty. Ltd., A.C.N. 000 237 598; and Brehmer Fairfax Pty. Ltd., A.C.N. 008 629 767; 180 Bourke Road, Alexandria, NSW 2015. *Learning with Your Computer* No 3 is printed by Macquarie Print, Dubbo, NSW.

The Australian Publication Emblem on the front cover of this magazine signifies our pride that the editorial content in this publication is produced and wholly edited in Australia, and that most of the advertisements herein are for products and services available within Australia.

Learning with Your Computer No 3 was compiled by **Kim Bucknole**, with production by **Donna Estrella**.

CONTENTS

Vol 3 **Page**

Buying a PC

Buying a PC is a straightforward decision, once you understand the jargon and eliminate the hype. Jake Kennedy discusses the steps in reaching a practical decision.

AT LEAST ONCE a day, I'm asked 'what sort of computer should I buy?' In slightly different forms, the question is asked by two distinct classes of users: those who are contemplating their first foray into the PC world, and those who want either a simple upgrade (replacing one PC with another more suited to the tasks currently at hand) or who want to expand an existing system by adding more computers. In every case, the advice I can offer is essentially the same.

Cost

THE FIRST concern of most users or users-to-be is cost; after is support, including warranty, and then comes performance. Buying a computer is no different from any other major purchase: a budget needs to be set, evaluated and re-thought, before any cash is handed over. If the PC is to be used as a business tool, it is essential to draw up a list of expected benefits, namely a quantified estimate of increased productivity and profitability. It's necessary to derive real *annual* dollar amounts: is the PC (or expanded system) going to save on office work; if so, what will be the savings in wages, either by doing with fewer staff or by not adding extra staff? Or, perhaps the computer is seen as a method of giving the business' principle time to get on with developing the business (after all, only accountants go into business to keep books, the rest of us want to practice *our* profession) – what is the value of new business likely to arise from the saved time? Even such nebulous reasons as 'improve the quality of life by spending more time with the kids' can be assigned a number for budget purposes, given some thought.

Once the PC's annual value to the business has been determined, halve the figure. If it's less than $6000, think about whether a computer is really necessary, or, are you not expecting enough from it? (That figure isn't arbitrary: $3000 is a good ball park figure for a basic business system and doubling it gives a workable basis for decision making. Besides, installing a computer is going to cause some disruption to the normal flow of work, so make it justify itself.)

Most small businesses which I've seen in this situation and then go on to purchase, come up with a figure between $7500 and $20,000. This is a good indication of the *most* you should spend, based on recommended retail prices. While a bit of shopping around will save up to 30 per cent on recommended retail prices, use the RRP in your calculations – staying on the conservative side is a good hedge against the unexpected.

The next step is to do a bit of research. Personally visit a number of dealers to familiarise yourself with the current offering, and with the dealership itself. Collect brochures, write down prices and ask what's bundled with the computer (that's marketing talk for 'what do I get for paying the price?' Right now, there are good software deals to be had here and many dealers are bundling printers at little or no extra cost.

Don't be afraid to pester the sales people with questions, no matter how dumb you think they are (take that either way). How a vendor handles 'dumb' questions is a good indication of its attitude towards customers and is a reflection of the type of after sales service you are likely to get.

Do not even think of buying a computer or monitor that has less than a 12-month warranty – if the supplier doesn't have faith in the machine, why should you? Twelve months is a minimum – many suppliers are offering two or three. Depending how heavily the computer system is to be relied, it can be worthwhile to pay for an extended on-site warranty – many of these include a replacement machine if yours is to be out of action for more than 24 hours. (Of course, that won't be of much use if no one has bothered to back-up yesterday's work.)

After several weeks of 'shopping' and asking questions, you should have a fair indication of how much needs to be spent. And – most likely, a fair amount of confusion. So – it's time to become more specific. Probably the biggest source of confusion to potential users is that there are a number of popular operating systems. The operating system is the link between the computer hardware and the software. Each software package is specifically tailored to run on a particular operating system – that's why Macintosh programs, say, can't be run on IBM-type computers.

Don't be afraid to pester the sales people with questions, no matter how dumb you think they are (take that either way).

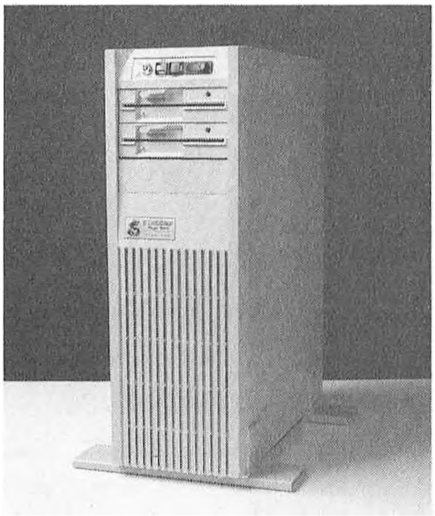
The most popular systems are those run by Atari's ST range (called TOS), Commodore's Amigas (AmigaDOS), Apple's Macintoshes (OS) and all the compatible 'clones' that grew from IBM's 1982-model PC, running MS-DOS (which stands for 'Microsoft disk operating system'). When looking around, you will come across several others: Digital Research's DR-DOS, an alternative to Microsoft DOS which has a number of features built into it that need to be added to the Microsoft system by buying utilities or 'fudging' (Mark Cheeseman compares the two systems in our December issue).

The current version of MS-DOS is 5 – settle for nothing less since it has memory management, an adequate word processor and file management built-in.

You are also likely to come across Unix and OS/2, but unless you can see the need for a multitasking or multiusing system, these aren't worth the extra hassle (it's a law of computer technology that the more power an operating system or software package has on offer, the more trouble it's going to be to get it to work – so keep things as simple as possible).

You will also hear a lot about Windows, which is 'shell' that sits between MS-DOS and software applications. Often referred to as an 'operating environment', it gives a common set of commands and 'cut and paste' facilities across programs that are written to be used with it. Depending on your particular needs, it may or may not be of interest. By all means check it out, but be warned that to take advantage of its features will take more memory, a faster processor and a larger hard disk than would otherwise be required.

Since the decision here is so basic to those that follow, let's back up a bit . . .



Even the selection of cases available for PCs can be confusing: floor-standing towers like the Syncomp pictured here offer plenty of room for internal expansion and don't clutter your desk, while smaller 'mini-ATs' and 'baby footprints' don't take up much of the desk, but generally don't offer much in the way of expansion either. It's important to get the choice right, since it's going to be very expensive to change it later.

What type of computer?

WHILE WORKING on the budget discussed above, certain tasks would have been assigned to be 'computerised'. For example, bookkeeping and sales contacts might be areas where increased productivity is anticipated – while visiting dealers, ask to see packages that do these tasks. Is there a single package that does both? (The less software that needs to be learned, the quicker the business will benefit.)

I'M LOOKING TO BUY A COMPUTER FOR THE HOME... PREFERABLY ONE THAT WILL BE GOOD COMPANY.
COMPUTER SPECIALIST
RING

Choosing

With recent developments in 'portable' technology, desktops are not the only choice for users: Inge Fuglestved discusses the alternatives.

ANYONE WANTING to purchase a personal computer is confronted with a confusing array of choices and options. The market is crowded with hundreds of different brands and models, all varying in speed, power, capacity and expandability. So what is the best way for a user to choose the right PC to meet their requirements?

Before deciding on a particular configuration, it is important to examine how the computer will be used. After all, it would be pointless buying an 80286-based machine to perform complex calculations or draw involved graphics. In the same way, most people would not buy a top-of-the-range '486 PC with a SuperVGA display if they only wanted to do some word processing.

Users should carefully consider their computing requirements before choosing the configuration they hope will best suit their needs.

Perhaps the first choice to make is whether to buy a desktop or a laptop model. Nowadays, many laptops and portables boast at least as much power and twice the versatility of their desk-bound cousins, but usually at a cost, because of the additional technology required to make laptops smaller and more robust. For those needing a computer which will travel with them, run on battery power or operate in places where the power supply can be unreliable, some kind of portable or laptop computer is a must.

These range in size from ultra-light notebook PCs through to the bulkier and usually more powerful portable computers. For example, Compaq markets a range of seven transportable PCs ranging from a basic 8086-based notebook through to the powerful SLT 386s/20 laptop. All offer users the ability to carry their 'portable office' with them from work to home, even to a hotel or sales presentations.

Those working in sales or journalism would probably choose a lightweight notebook computer which fits neatly into a briefcase or tucked under the arm. These form the largest growth segment of the computer market. They are usually about the size of an A4 notebook (hence the name) and weigh around 3kg. With battery power, liquid crystal display (LCD) and a choice of processor (8086, '286 or '386SX), notebooks are highly versatile tools for calculating spreadsheets, writing and note-taking, one-on-one demonstrations and off-site programming.

For those wanting the benefits of portability without the heavier price tag, a slightly larger laptop or portable computer might prove an ideal solution. More powerful portables are popular with accountants, business executives, researchers, engineers and architects who need to carry their work with them from the office to visit clients.

*The key to making the best purchasing decision is ensuring that current requirements are met, while allowing sufficient expansion capability for new software developments that will further increase productivity and quality – **Inge Fuglestved, marketing director, Compaq Computer Australia.***

If portability is not an issue, then the programs the user intends to install will become one of the main criteria in deciding the final configuration. Arguably, the standard in business software today starts with Microsoft Windows, the graphical user interface which has sold millions of copies around the world. Since Windows requires a '386SX computer with at least 2Mb RAM to run at op-

The software has a significant bearing on the configuration of a machine – a PC that can just run the basic tasks mentioned above, will be quite a different beast from one used for graphic design work, musical composition or text retrieval from a massive database.

Now, back to the 'which' question. If you use a PC at work, it makes sense to buy a similar machine for home (and vice versa) if similar applications are going to be run in both places. Another common reason for buying a computer is for the kids to use, either for recreation or education – it's then a good idea to buy the same type of computer that is being used at the local school.

It isn't so much that the kids will be familiar with the machine, it's that they have a resource in other users that they can turn to for advice or help. The same logic applies to a computer for use in business – if you buy a brand new software package, that no-one else has ever heard of, who are you going to turn to when it won't work with the printer?

If games and animation are the prime interest, or DTP a major requirement, then different machines will suggest themselves as you shop around.

For any new user, an IBM-compatible PC, and its various compatibles and variations, form a group which is hard to ignore. There is a wealth of software available for these machines, commercially and as public domain (free) and shareware (pay if you use it). Business applications are particularly well represented and education, graphics and music software are becoming more sophisticated.

On the topic of public domain and shareware software – these are an excellent source of software if you want to try a certain type of application. Public domain software is free, while shareware asks that the user register the copy of the software with the author if it is going to be used. For example, you may have decided to buy an accounting package, but aren't sure if you need a spreadsheet. There are a number of excellent ones in both categories, many of which can exchange data with the full-blown commercial packages. Back to the hardware . . .

The IBM and its compatibles may not cover any particular area of activity as well as more 'specialised' machines, but they tend to be the jack-of-all-trades, able to cope competently in most areas. As an example, our minimum recommendation for a small business system is a 40Mb hard disk, a floppy drive, 1Mb of RAM, a colour monitor and a letter-quality printer with a

the right PC

timum performance (although it will run *adequately* on a '286), this should be considered the minimum configuration for those wanting to run this 'graphical user interface'. Those requiring a full suite of business software such as Windows, a word processor, presentation graphics package and spreadsheet program should definitely consider a '386SX, a full '386, or even a '486 computer, although the latter is really only necessary for power users who work with lots of graphics, computer-aided design (CAD) or *very* large spreadsheets.

Of course, if complicated calculations and spreadsheets comprise a large part of the workload, a maths co-processor can be added for enhanced performance at minimum cost.

It is also important to plan for future needs when buying a PC. The latest software products released onto the market offer far more features and, as a result, are considerably more memory-hungry than their predecessors. Therefore, choosing a configuration which only just meets current requirements could limit any possibility for growth further down the track. For those planning to buy a computer for word processing only, a '286 will probably be adequate for their needs. But they should consider whether they are likely to increase their demands over the life of the computer.

The average business PC has a life of perhaps three to four years before it is considered obsolete, although the rate of technology change dictating this period will reduce rather than increase. While two years ago, the average business PC was a '286 machine with perhaps a 20Mb hard disk drive, the trend has now moved to favour '386SX and '386 computers with larger capacity drives. This has been sparked to some extent by the move in some circles towards more powerful operating systems such as Unix, OS/2 and Windows, but today's standard business packages are also becoming more demanding.

Considering that some applications take up several megabytes of storage space once they have been installed on the hard drive, then a suite of business software may account for well over 10Mb, even before you start saving any files. In that situation, 20Mb starts to look a little inadequate. Most high performance business machines today offer a minimum 40Mb drive with options for up to 200Mb or more.

With the trend towards network computing, the personal computer has really come into its own. Instead of tapping into a super-powerful mini or mainframe computer through a dumb terminal, the business users maintain their software on the hard disk of their PC, using the network for file storage, communications and sharing peripherals like printers, tape backup units and uninterruptable power supplies. In some cases, the applications are stored on a powerful fileserver, which is then also used for maintaining critical databases and files.

With a powerful, high-capacity fileserver, it is possible to by-pass the need for large internal hard drives on the individual workstations. For example, Compaq's DeskPro 386N series, designed for network use, offers built-in locks and other security features, expansion slots for network interface cards and communication boards, as well as the option of diskless models for maximum security.

Perhaps the best of both worlds is Compaq's LTE 386s/20, a notebook-sized '386SX offering three hours battery life, which can be configured with a desktop expansion base, external colour monitor and plug-in keyboard. The LTE 386s/20 is powerful enough to run any of today's business software, provides up to 60Mb internal storage and can connect into any standard LAN via the desktop expansion base. It offers all the benefits of lightweight portability with no compromise in power.

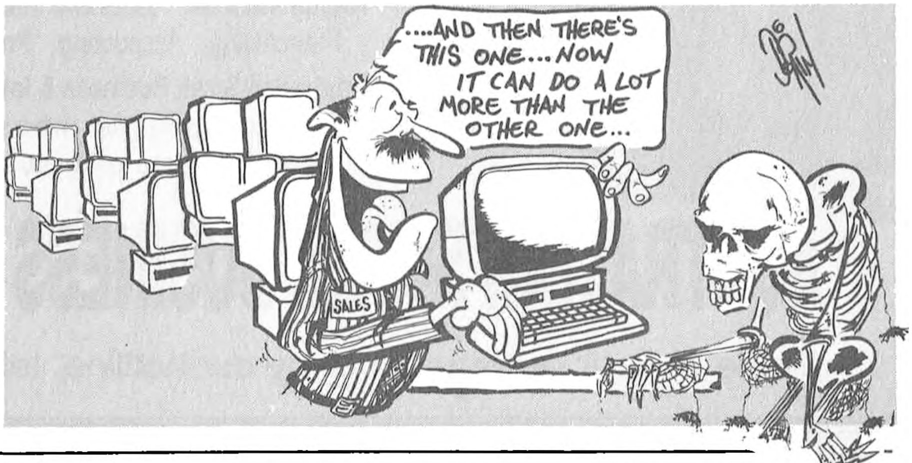
Once the computer's configuration has been chosen, the user must decide which brand they want to purchase and where to buy. To avoid problems with poor quality or incompatible products, it is always advisable to buy a PC made by a manufacturer with a solid reputation for reliability and value.

Recognised name products might cost a little more initially, but the user will save in the long run by avoiding downtime and technical problems. Buying from a reputable dealer will also guarantee competitive pricing and reliable, expert support in the event of any technical problems.

There is a PC on the market today that is just right for all computing requirements. The key to making the best purchasing decision is ensuring that current requirements are met, while allowing sufficient expansion capability for new software developments that will further increase productivity and quality.

draft mode. (If that sounds too technical, you've got the first half-dozen questions to ask when shopping.) In the IBM world, such a system can be had for well under $2000 – for basic business applications on the other types of machines, the prices will be about the same.

The archetype 'personal computer' – the PC – was the original IBM PC, running a 4.77MHz 8088 processor. The 8088 is a microprocessor chip that functions as the central processing unit (CPU), the 'brains' of the computer. Since the PC, IBM and the clone makers have released the 80286-based AT and the 8086-based XT. The company that developed the chips, Intel, has since released faster and more sophisticated members of the family: the 80386DX and a cheaper, less fully-featured version, the 80386SX, the 80486DX which has a maths co-processor built-in – this is important if you will be working with calculation-intensive applications such as large spreadsheets or CAD (computer-aided design) – and the 80486SX, a stripped down version without the co-processor, but still offering about 25 per cent more processing power than a '386DX. XTs have been discontinued by most manufacturers and


...AND THEN THERE'S THIS ONE... NOW IT CAN DO A LOT MORE THAN THE OTHER ONE...
SALES

ATs are going the same way (in the latter case, there are real bargains to be had, if a '286-based machine suits your needs).

If most of your work is text processing or simple accounting, an AT will do the job. However, the '386 machines are now in the same price range, so it is probably a good idea to check both out. All technicalities aside, the most meaningful way of determining if a system is what you want is to use it – ask to be allowed to try a computer in the showroom for half an hour or so; after you've done this with three or four different computers, the differences in speed will become apparent (or not, depending on your needs).

Under the cover

PERSONAL COMPUTERS all have a traditional form: a box on the desk (the 'system unit') with a monitor (display) on top, driven by a keyboard, and perhaps, a mouse. The system unit typically has a sheet steel case, with a cover that either slides off after a few screws are removed, or has a lid which can be tipped up to reveal the innards.

The most noticeable object under the cover is usually a very plain looking box, the power supply – before buying ensure that it is heavy-duty enough for your future needs. If you are likely to add a larger hard disk or other bits and pieces that draw from the power supply, make sure you buy a machine with over 200W or more available: much less and you'll be replacing it at some stage. The other sizeable hardware bits of interest are the disk drives – it's worth noting how many 'devices' (additional floppy drives, hard disks, tape backup units, whatever) can be added before the case is full.

All technicalities aside, the most meaningful way of determining if a system is what you want is to use it.

Flat across the bottom of the unit, usually running from front to back at the left side, will be a large printed circuit board full of electronic components – the motherboard. This is the home of the CPU and where the ubiquitous (optional) maths coprocessor plugs in. Daughterboards for specific applications such as video and hard disk control plug into the motherboard, sticking up at right angles to it. Next to any daughterboards (there may not be any if all functions are built into the motherboard), will be at least several slots (sockets) into which other boards plug. These are used to add the likes of enhanced graphics, additional RAM or serial ports, and internal modems and fax cards.

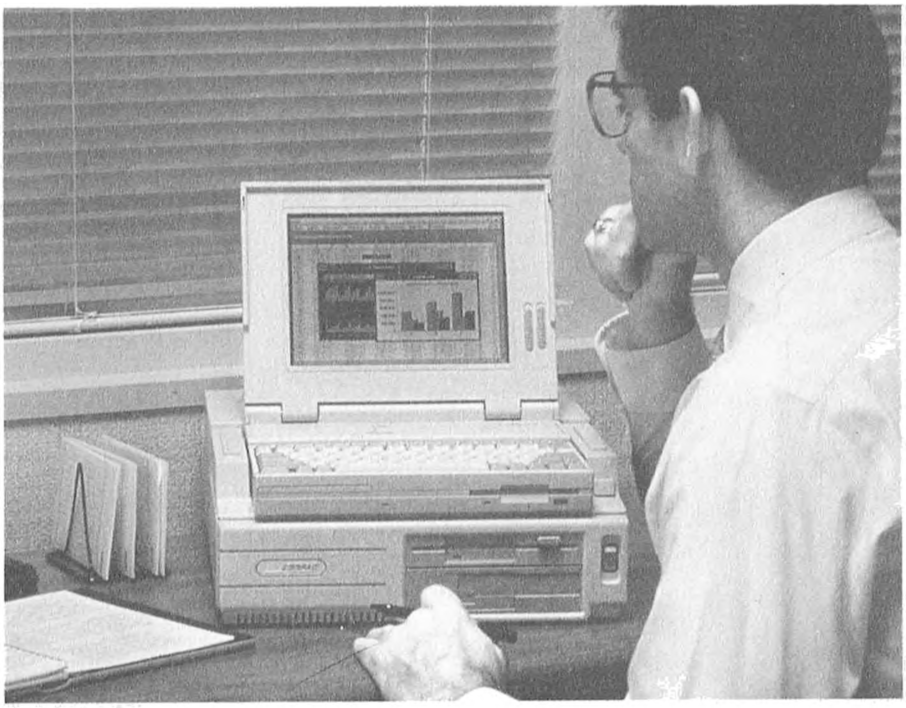
Brochures and advertisements often claim something like 'eight expansion slots'; before using that in your decision making, check to see how many of these are free (unused). Also, there are 'mini' and 'baby' desktop units, which occupy less desktop space, but – be warned! – many of those cannot physically accommodate standard add-on cards.

While looking under the hood, note how neat the arrangement of cables is and to

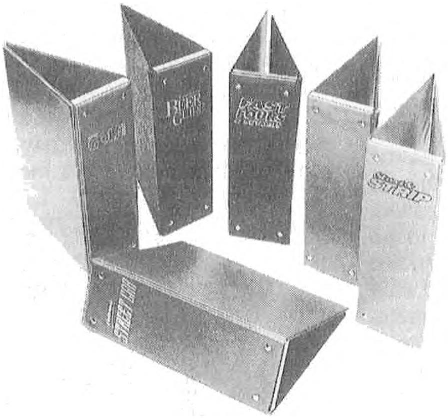
what extent surface mounted chips have been used on the boards (generally that means, fewer components sticking up in the air). As a rule of thumb, the neater a PC looks on the inside, the more reliable and easier to expand the system will be.

Memory

ANOTHER AREA that frequently causes confusion is memory. There are three types of memory in a computer: ROM (read only memory – it needn't concern us here, since it won't generally come into the purchasing decision), RAM (random access memory, which is important) and 'magnetic media' such as floppy and hard disks, and tape drives. All software requires a certain amount of RAM to run. A standard AT or '386SX is usually fitted with 1Mb of RAM (or 'memory', as it's often called). Now, after the operating system – just another program, really – is loaded, there might be 300Kb of that left. So, on this machine, you cannot run software that needs more than about 500Kb of RAM, or a combination of software that needs more than that at one time. Software today tends to be quite full-featured, but that also means that it needs a fair



A portable with an expansion box – like this Compaq LTE – can offer a cost-effective solution if you need to work on the move as well as in the office.



Reset and power buttons on the front panel, a full set of informative LEDs, VGA graphics, 1Mb RAM and a 40Mb hard disk – this Samsung '386SX is an example of the configuration you should be looking for.

whack of RAM to show off those features, and for any applications, 500Kb isn't much.

RAM (also called 'conventional memory'), extended memory, expansion memory – this area is probably one of the most confusing to users. (Stewart fist gave a comprehensive overview of computer memory, 'Extended, expanded . . .' in our May 1990 issue: back copies are available for $6 from the Office Services address on the Contents page.

Memory is the main reason you should determine the software you are going to be using before buying a computer – there are few things more frustrating than buying a software package that's all you've been looking for, only to get a 'not enough memory to run this application' error message when you fire it up.

At some stage, every user needs to connect external devices (peripherals) to their computer, whether it's a printer, a mouse or a modem. These are connected by plugging cables into ports (the sockets on the rear edge of an expansion card, poking out through the opening in the rear panel). The most common configuration in an IBM-type computer is one parallel and two serial ports. At this stage, never mind what 'serial' and 'parallel' mean, just remember that you will need a parallel port for a printer (although some use a serial port), while serial ports are used for modems, mouses and a variety of other external devices. (Many machines are available with a 'bus mouse', which means the mouse port is built into the motherboard, so the mouse doesn't take up one of the serial ports.)

The one/two combination is most probably all you'll need to begin with; there are quite inexpensive boards with additional serial ports that can be added if you need them later.

Often the serial and parallel ports will be on the same expansion card along with other functions like memory and the clock/calendar. These combination boards will be called multi-function or multi-I/O boards, but what is on them will vary from one to the next.

Software

A COMPUTER without software is useless. The most common software needs are for a word processor, database and spreadsheet. There are a number of integrated packages that supply all of these as one 'suite', or offer various other application mixes. If they suit your needs, these can be good value, not only in dollar terms, but in ease of learning since the applications will share a similar 'user interface' – that's jargon for how a program looks when you are using it and the actions necessary to accomplish tasks like entering data and saving files. These can include various key combinations, commands and different ways of using a mouse, some of which you will quickly discover are almost intuitive, while others will never seem natural.

Many suppliers bundle a selection of software with computer systems, so keep that in mind when shopping around – it could save some money *if* it's the software you need. If it's not, ask for a different bundle with the same total price – while many dealers aren't able to do that, some can. If you buy a collection of packages from different software manufacturers, check how compatible they are – can your spreadsheet accept data from your database, for example?

When you have decided to buy a PC, remember that there will be a few necessary extras. Initially, you may be able to avoid buying a printer, but, inevitably, you

I THOUGHT YOU WENT OUT TO BUY A COMPUTER?
I DID... BUT, AFTER BUYING THE MAGAZINES AND BOOKS ON BUYING A PC..I HAD NO MONEY LEFT.

will find you need one. Good quality dot matrix printers are now under $500 and others can be had for $300 – they might not give 'letter quality', but the results are perfectly adequate for many uses. When you buy that printer, make sure you get a cable to suit – for some reason the cable is often a $30 'option', even though the printer is of no use without it!

Even without the printer, you will need some disks, and less than about 30 of each size won't be enough, besides, buying disks a hundred at a time will often attract a discount. Paper and ribbons for the printer will be needed. A box of paper will cost between $50 and $80, and you should always have one spare in addition to the one in use. Ribbons always wear out at the wrong time, so have at least one spare – at around $15 each for most printers, the expense is not too great. Before settling on a printer, ask about the cost of replacement ribbons: it can be as much as $70.

Hard-where?

ARMED WITH a list of your needs (don't worry if they seem loose to begin with, as you look around, you'll soon tighten them up and fill in the details), look through the ads in the magazine, your local Yellow Pages, and the computer sections of local newspapers. To get a feel for things computer-ish, spend time in a newsagency with a good selection of computer magazines – beware though, that most of these magazines are from overseas and the information in them is usually out of date and not relevant to Australia. That aside, they are still an excellent source of general information on the capabilities of the various machines, particularly those with the proprietary operating systems mentioned earlier, and of the range of software available.

You should then be able to draw up a short list of software that meets your specifications and of the hardware that can run the software. Now, go back to three or four computer stores – the bigger, the better. While you might get a better price and more personal after sales service from one of the smaller retailers, while you are shopping around, you want to be able to see the largest range of products you can – so, visit the big dealers, and once you have made your decisions, also approach two or three smaller dealers and ask them to quote on a system.

Before proceeding any further, it's time to talk to your accountant. Buying isn't the only option: like cars, systems can be rented or leased. There are also a number of other areas that an accountant can advise on to make sure you get the maximum return from your investment. In fact, if you lack confidence, it's a good idea to get your accountant involved right from the start.

WE'RE NOT SELLING COMPUTERS TODAY...BUT, IF YOU BUY ALL THE ACCESSORIES WE'LL GIVE YOU A FREE ONE...
SALES

More helpful advice can be found at local User Group meetings – we can't stress enough what a valuable source of information and expertise these groups are. Everyone there will have been a new user at one time or another and you'll probably hear a number of tales of the pitfalls to be avoided – you'll soon learn who the good and bad companies to deal with are. User Groups generally advertise meeting times in local papers and dealers often know about the local groups.

Initially, you might even want to take a computer-literate friend or business associate along when you are visiting dealers. Don't rush your decision – take your time, compare prices, quality and advice. Let things stew for a few weeks and then think about the choices you want to make again – if you are feeling confident, it's time to move.

Motherboards, 32-bit, HGC, megabytes – it all sounds unfathomable to newcomers. But don't be daunted: the only way to learn is to start asking questions and having a look around. A PC system represents a sizeable investment for anyone's business. Take your time, ask even the dumbest sounding questions (walk away from condescending replies), don't just look at computers from different manufacturers and the software that runs on them – *try them!* Small things like the feel of the keyboard and presentation of software error messages will make a difference to how happy you'll be with your purchase in six months. □

IMPORTANT

Previous issues of *Learning with Your Computer* have been exceptionally well received. There are no back issues of *Learning with Your Computer* Nos 1 and 2 available.

The material appearing in *Learning with Your Computer* No 3 has been published previously in *Your Computer* monthly magazine. Accordingly, where prices for products are given, they provide an indication only, and you should check current pricing and availability with your dealer or the distributor of any product in which you may be interested.

***Learning with Your Computer* No 4 is scheduled for publication in November 1992, and will contain articles dealing with multimedia, mass storage, CD ROM, optical storage, Forth, C++, and random numbers, as well as the last seven parts of the series on compiling QuickBasic.**

Start Computing!

- Part 1

Computers are useless unless they can be told what to do by programs – let's start computing with an overview of programs and the languages in which they are written.

A COMPUTER AS purchased is simply a set of hardware – lots of transistors, large-scale integrated circuits, disk drives, a keyboard, display screen and perhaps a printer. This equipment in itself is totally useless; it cannot go, and it wouldn't know where to go even if it could.

The hardware you purchase is only useful when it has a program running it. We can define a program as *the specification for using the hardware so that it can convert input data (within the domain of the program) into output values (within the range of the program).*

The 'domain' means all the kinds and sizes of input data acceptable to that program. This may include numbers, symbols, functional notation, pictures or perhaps all four. There is usually some limit imposed on the largest and smallest numbers which can be handled. A few programs can take monochrome and colour pictures as input data, some only monochrome, most programs cannot handle pictures as input at all.

The 'range' means all the kinds and sizes of output which that program is capable of providing. Output possible from a particular program may be restricted to numbers and headings, with a limit on the possible sizes and types.

The above description was not always the case. The earliest computers had inbuilt hardware programming, meaning that each computer was physically wired to do a single task. Today we would call that wired logic – it's still found in some older type of dedicated scientific measuring equipment such as averagers, histogram drawing instruments and live signal integrators. Though inflexible in use, the wired logic idea leads to fast processing.

The primordial hard-wired mathematics machines designed to solve a single problem, or today's analog computers whose program is in the pattern of connecting wires between circuit sections, both fall

__Start Computing__ was written by Bryan Maher, a freelance technical writer and formerly senior lecturer in computer science at the Capricornia Institute of Technology, Townsville Qld.

into the same hardware programming class. ENIAC, built in 1942, was the first electronic digital computer; it had to be rewired each time the user wanted to solve a different problem.

It wasn't long before people sought a better way to program computers. In 1945 John Von Neumann proposed and inspired EDVAC, incorporating the revolutionary idea of storing the program in a section of the computer memory. Thus was born the modern idea of software programs as digital electronic signals read into and held resident in a section of memory.

A program generally consists of at least two parts –

The Operating System which tells the computer how its various components should interact and how to interpret commands, for example, how the electrical signals generated by pressing the Return key are to be read and acted upon by the machine; and

The Program itself which is used to define the problem you want solved, or the game you want played, this part of the program can be changed at will, either by altering the program itself or by changing to another program.

In larger computers, there are usually other program parts resident in memory (on smaller computers they are read in from the disk when needed). These are the service programs intended to assist users; the common ones include –

Editor programs to enable programs and data files to be written and corrected (debugged);

Translation programs such as various Assembler, Compiler and Interpreter programs (more on these below);

Library programs which hold instructions on just how to perform various operations;

Linker programs, used by the computer to join together sections of different programs as required, for example if you write a program calling for, say, the square root of some number, the linker searches the library, finds the section which describes how to calculate a square root, and attaches (links) a copy of that section to the program; and

Communication programs which are used to make one computer talk to another.

Alogorithms

THE MOST basic concept in programming is the algorithm – a series of step by step instructions that produce a result (academics add to that definition the phrase 'within a finite amount of time'). Any set of instructions, whether assembly instructions for a kit or a recipe for a cake, is an algorithm. The basic algorithm for developing a program is –

1) Define the problem;
2) Plan the solution;
3) Write the program;
4) Test and debug the program; and
5) Document the program.

Each of the above steps can, of course, be broken down into its own algorithm. To make algorithms easier to follow, they are represented graphically in a flow chart.

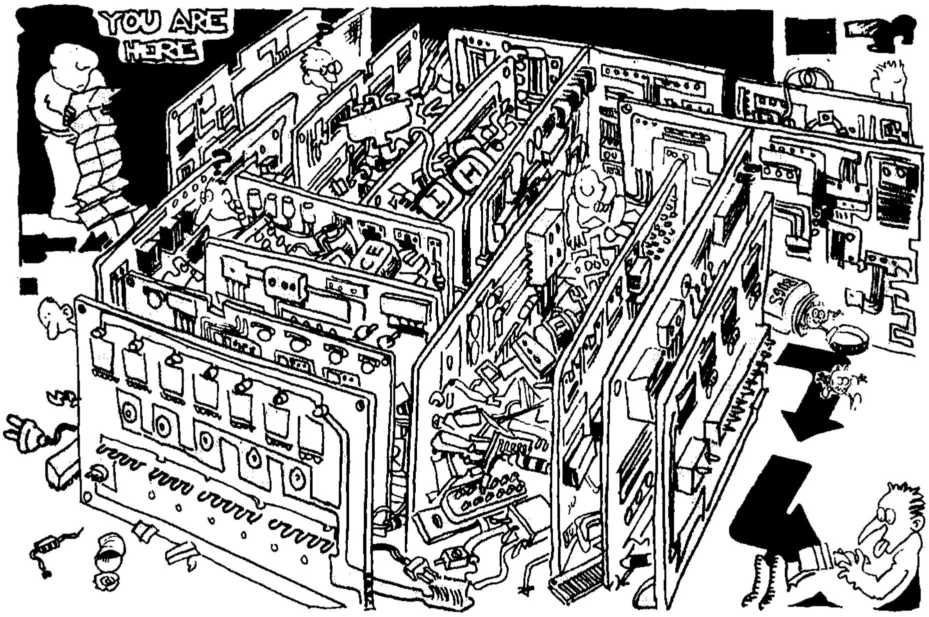
Spoken and computer languages

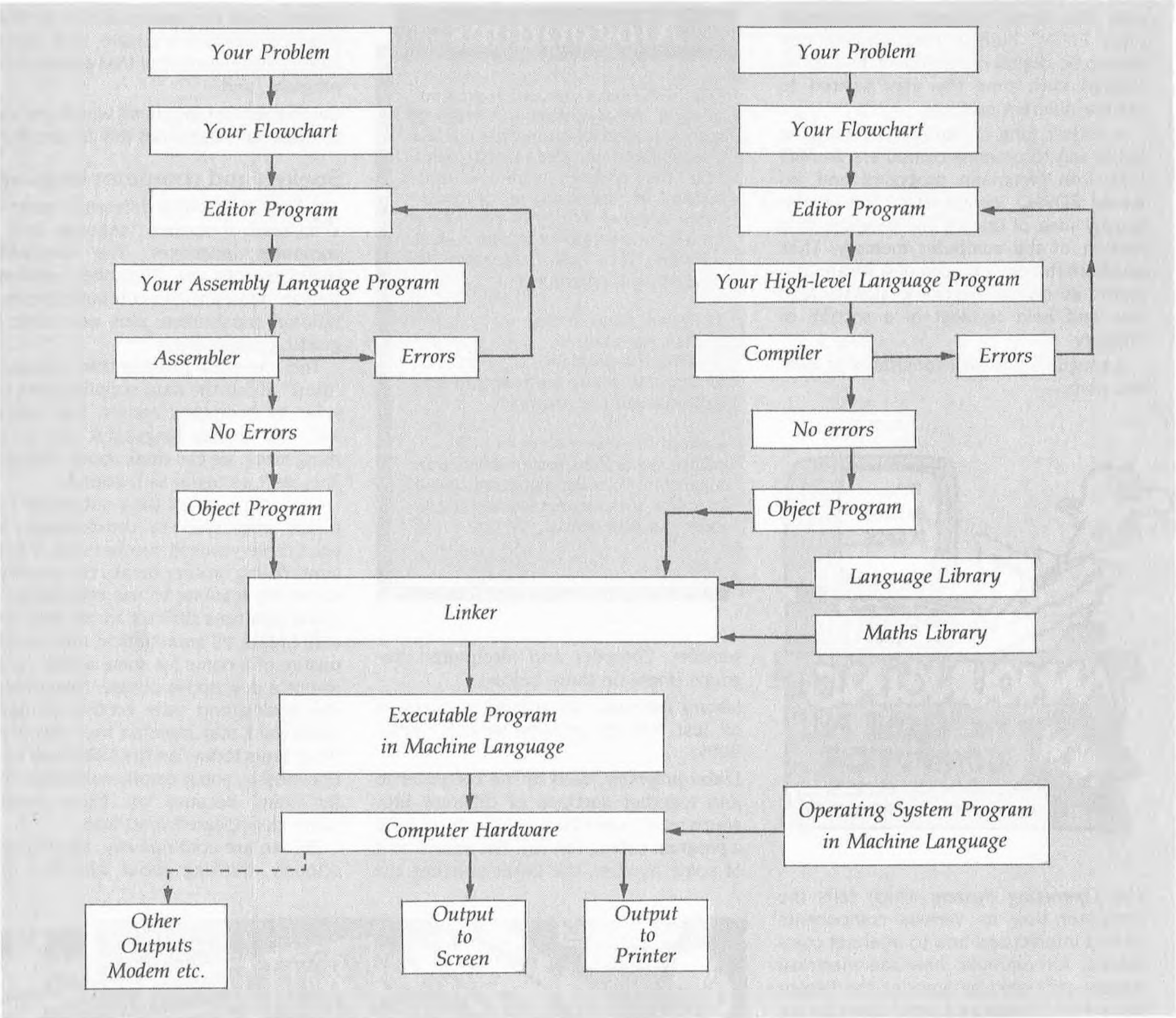
A FUNDAMENTAL difference exists between a spoken language and all computer languages. The comparison boils down to the differences between a human mind and the computer's computational capabilities, plus one basic linguistic consideration.

The cleverest programmed computers 'think' about the data supplied, and consider its computed results. But humans, with our spoken languages, can do one thing more: we can think about 'thinking'! This isn't as trivial as it sounds.

Consider yourself (as a substitute computer) engrossed in conversation with your three-year-old brother and a great-aunt. Young brother speaks two sentences about his learning to use telephones; he mispronounces difficult words and probably breaks all grammatical rules; if he is unsure of a name for some article, he will invent a descriptive phrase. Nevertheless, you understand little brother perfectly.- Great-aunt may mention the unavailability of seats today 'on the 480 Bondi tram'. Effortlessly, you mentally substitute 'bus' for 'tram' because you know Sydney's trams disappeared years ago.

So, we are continuously, albeit unconsciously, thinking about what the other





Block diagram of 'software paths' in a computer.

person is thinking. Armed with our background knowledge, we become an intelligent interpreter. Now – how to invent a computer language to do likewise?

Forgiving languages

TRUE, SOME computer languages are very forgiving of our mistakes and sloppiness. FORTRAN, for example, doesn't mind whether we write –

```
        GOTO2
or      GO          TO          2
or even GOT    02
```

That language ignores most spaces, so to FORTRAN all three instructions, mean the same thing.

Yet FORTRAN will disolve into error if you write that GOTO in column 1, but most versions of Pascal or PL/1 wouldn't mind at all. However, no computer language has been invented which could cope with your baby brother's wrong grammar (syntax mistakes), his mispronunciation, or spelling mistakes.

Could some computer language correctly interpret exactly what great-aunt's 'tram' meant? Simple substitution of 'bus' for 'tram' is insufficient because at some stage she may well mean 'tram'. For a computer to make the appropriate substitutions, it would need to know aunty's (and much of humankind's) history.

So, to communicate with a computer we must learn to speak a language it understands – the best way to start learning a computer language is to learn about the very basic aspects of the computer.

The computer

THE COMPUTER is a binary digital machine (all that means, really, is that its 'thoughts' are all built on combinations of only two numbers, 0 and 1). It consists of many electronic components, each having an output signal voltage which is always either low (0) or high (1), never in between.

For the components used in the IBM-type personal computers, and many others, the low level is about 0.2 volts, while the high level is about 4 volts. These voltage levels often switch at millions of times per second, in strings of pulses. Each pulse, and the information is carries, is called a 'bit' (BInary digiT).

Communication from keyboard to computer, and from computer to screen is achieved by groups of eight bits sent serially (one after the other). Each group of serial bits is called a 'byte'. The signal initiated when you press a keyboard key is a byte of eight symbols, each of which is either 1 or 0. For example 01011010 is a keyboard byte sent to the computer; this particular byte representing the upper case character Z. There are 256 possible combinations in this 8-bit byte – that's enough combinations to uniquely symbolise each of our alphabetical, numerical, control and special characters in the extended ASCII (American Standard Code for Information Interchange) code; extended means all 256 characters while standard ASCII code is the first 127.

Some hardware, particularly printers, use eight parallel wires to carry all 8 bits of a byte simultaneously, rather than serially.

Within most computers, as more than 256 combinations are required, two or more bytes are joined together to form one longer 'word'. In the IBM PS/2 Models 30, 50, and 60, also in the earlier XT and AT, each computer word consists of two bytes joined, that is, the word consists of 16 bits – so, these are known as 16-bit computers.

Word length

WHILE MANY PCS on the market are 16-bit machines, a few are only 8-bit; the Commodore 64, for example. However, some such as IBM's PS/2 Model 80, or other machines using the 80386 microprocessor chip, join 4 bytes (32 bits) together to form each word – thus they are known as 32-bit computers.

The next size above these, collectively termed micros, are minis, which are usually 16-bit machines. Examples are the Hewlet Packard HP-3000 and HP-1000 series, and the DEC PDP-11 while the earlier DEC PDP-8 was an 8-bit minicomputer. (Here, we are only getting the terminology clear – why a mini is a size up from a micro and so on, we'll leave for a later discussion.) The term super mini describes larger-than-mini machines such as the VAX-750, which use a 32-bit word.

Larger still are mainframes, those monsters which fill a large room with big cabinets surrounded by high-speed tape drives machines and multi-layer, fast-access hard disks. For many years these machines have been using 32-bit words; an example is the IBM 360 series which dates from the early 1960's.

Other mainframes use 36-bit words (the Unisys UNIVAC-1100/42) and the giant Control Data Model 6000 uses 64 bits to make up every word.

The computer is sensitive only to voltage signals which are most commonly generated by means of the keyboard. Other methods include a mouse and joystick. Both of these are mechanical methods for driving a potentiometer to produce the required voltage signals.

The fastest input method is for the computer to read the magnetically-coded information on a disk, whether it's a fixed (hard) disk, a floppy disk or a compact disk. From earliest times, computers have electrically read the magnetically coded program and data on magnetic tape; this medium is slower to read than a disk, but it's capable of storing enormous quantities of information.

But – we were talking about computer languages: how are these voltage signals related to a language? Let's look at the language that's at the heart of every computer – machine language.

Machine language

A TINY SEGMENT of a program, using the high/low, 1/0 convention, for a 32-bit machine might look like this –

```
101001110000011011100101111000011
01011000111011100011000111100100
00000111111011001000110110110011
```

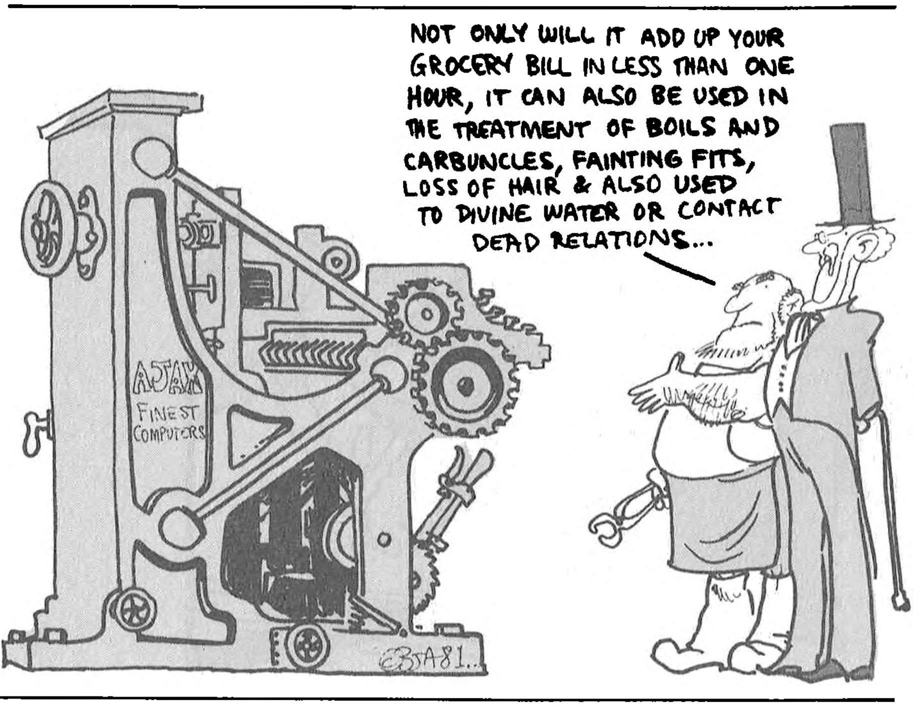
where the 1s and 0s tell us the voltage levels on each of the 32 parallel lines.

If this segment were describing the contents of memory, it would be showing the output voltage levels of each individual memory cell that make up the group of 32 cells (a memory address) storing one 32-bit computer word. Now, even for a small program there may be perhaps 50,000 words to write (and understand)! That is certainly not the way we want to talk to, or program a computer.

But, as we've seen above, that's the only language the machine understands, so it's termed a machine language. Machine language is used to give the computer's central processing unit (CPU) basic instructions like 'add' or 'shift right'.

The complete set of instructions understood by a particular machine's CPU is called its instruction set. To write programs in machine language you would need to use all those instructions and the binary word corresponding to each.

Nobody wants to write programs in machine language, but they must be produced somehow since that's what the


NOT ONLY WILL IT ADD UP YOUR GROCERY BILL IN LESS THAN ONE HOUR, IT CAN ALSO BE USED IN THE TREATMENT OF BOILS AND CARBUNCLES, FAINTING FITS, LOSS OF HAIR & ALSO USED TO DIVINE WATER OR CONTACT DEAD RELATIONS...
AJAX
FINEST COMPUTERS

computer understands. As you can imagine, early programmers soon sought to make things easier.

Assembly language

THEIR FIRST improvement was to write machine language programs in numbers with 16 as a base, instead of 2 (binary numbers). This made programming easier, but it was still difficult and taxing.

But humans, with our spoken languages, can do one thing more: we can think about 'thinking'!

Their next improvement was to devise Assembly language which uses cryptic code like MOV (meaning 'move the data') and MVI (meaning 'move immediate') and expresses all numbers in hexadecimal form. Having written their program in that form, it was then necessary to 'assemble' it into equivalent machine language. Though assembling by hand is possible, that's slow, tedious and prone to error, so they wrote an assembler program to do the translation for them.

Unfortunately, different microprocessors have widely differing instruction sets and machine language equivalents. Therefore, a different Assembler program must be written for each different microprocessor chip type. Furthermore, as Assembly language itself must know the instruction set of the machine, it follows that there is a different version of Assembly language for every type computer – that means that Assembly language programs are not portable.

Machine Language, written in either the 1s and 0s of binary or in hex, is the base level or lowest level of programming – low level means right down at the level of the machine itself. The highest conceivable level would be programming in English (if that could ever be attained).

Assembly language is one level higher. At this level, the language has intimate control of the computer, that is, it can invoke any (or all) of the operating system commands. But, with that privilege comes responsibility: Assembly language programs must spell out every instruction in minute detail. For example, if the requirement is to add X and Y together, not only must the ADD instruction be given, the system must be told whereabouts in memory X and Y will be found, then (in absolute detail) just how to perform the addition. When the addition is done, the computer must be told at what memory address to write the sum.

Despite this detailed requirement, Assembly language is still used because it gives the most concise and fastest-running programs. Also, as Assembly language can access every operating system command, graphics can be programmed in detail (limited only by the capabilities of the machine and display). Many of the video games machines in in arcades were originally programmed in their own Assembly language.

But, because of the detailed instructions required, Assembly language is still very slow and tedious to program.

JOIN A COMPUTER CLUB TO MEET OTHER USERS & GAIN EXPERIENCE.....

FORTRAN

IN ONE tremendous leap in 1954, programmers at IBM invented a new Computer Language called FORTRAN-4 (for FORmula TRANslator 1954). Originally written for one large IBM machine, this language evolved, was extended, then run on other IBM models. It was the first high-level language.

The IBM programmers had two aims in writing the language – first, they wanted to produce a computer language much closer to English than Assembly; and second, they wanted it to be portable so that it would run on any machine.

Of course programs written in FORTRAN still had to be translated to Machine language, so a a translation program called, a FORTRAN Compiler, was written. The Compiler first checks the FORTRAN program for errors, then translates it into machine code. (Programs written in a high-level language are referred to as source code – they are the 'source' of the machine language for the program.)

The Compiler includes two extensive libraries. The first, the FORTRAN library, contains detailed machine code instructions needed to implement the operating system commands, such details as where in memory things should be placed, exactly how multiplication is to be performed, how data is to be read in, when and in what form results are to be written out. The second library, the Maths library, contains machine code routines for the calculation of many mathematical functions. In fact, FORTRAN and its Compiler was heavily biased towards high-speed solutions of involved mathematical equations.

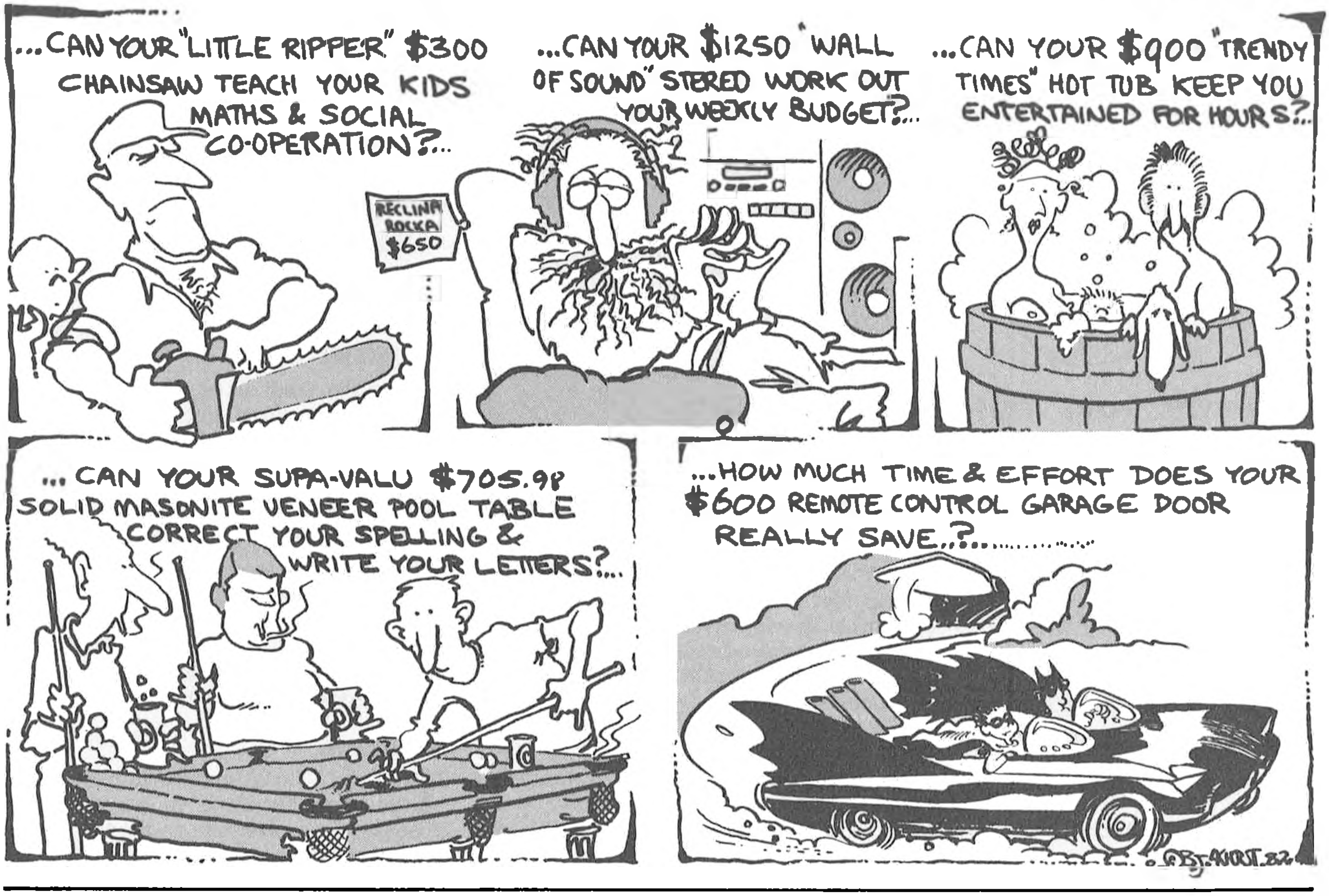
Because the Compiler intimately invokes the operating system and its instruction set, for each different model of computer and each different operating system, an appropriate FORTRAN Compiler must be designed. The whole point of any high-level language like FORTRAN is that the same source program can be fed into any machine, the Compiler for each machine (and operating system) rewriting all source language instructions into the corresponding machine language commands peculiar to that machine (and operating system).

To achieve this machine-independence, the designers excluded all operating system commands from FORTRAN; for example, CLS (Clear The Screen) command in that language.

Programmers of the time used it to wrestle with long involved calculations, tasks such as the US Presidential elections results, statistics from the National census, trajectory calculations for the Defence Department, engineering, scientific and mathematical calculations never before attempted.

The designers of a language can include within it any functions they choose. Therefore, they included within FORTRAN all the common mathematical functions, plus logarithms, exponentials, complex arithmetic, trigonometrical functions, even one of the hyperbolic functions. The appropriate Maths library routine usually calculated each function from its mathematical defining equation, often as a truncated infinite series.

How To Justify Computing . . .


...CAN YOUR "LITTLE RIPPER" $300 CHAINSAW TEACH YOUR KIDS MATHS & SOCIAL CO-OPERATION?..
RECLINA ROCKA $650
...CAN YOUR $1250 "WALL OF SOUND" STEREO WORK OUT YOUR WEEKLY BUDGET?...
...CAN YOUR $900 "TRENDY TIMES" HOT TUB KEEP YOU ENTERTAINED FOR HOURS?..
... CAN YOUR SUPA-VALU $705.98 SOLID MASONITE VENEER POOL TABLE CORRECT YOUR SPELLING & WRITE YOUR LETTERS?...
...HOW MUCH TIME & EFFORT DOES YOUR $600 REMOTE CONTROL GARAGE DOOR REALLY SAVE..?..........

To further increase running speed, the fathers of FORTRAN designed program loops for high speed execution. They concentrated their efforts on the design of fast-access Block Structures, and the high-speed DO LOOP. Indeed, even today the FORTRAN language DO LOOP is the fastest type loop available in any language. For these reasons, 95 per cent of all engineering, scientific and mathematical programs on sale worldwide in 1988 are still written in FORTRAN.

Properties of FORTRAN

ASIDE FROM those properties mentioned in the text, other charcteristics (and limitations of) FORTRAN include –

□ Very little punctuation is required;

□ There are no reserved words;

□ It can be written, tested and compiled in sections, called subroutines or function subprograms, facilitating the production of very large programs, each subroutine is independent which means they are not nested;

□ While it is poor in equation manipulation, it's rich in equation solution;

□ Of the different methods of looping within a program, the FORTRAN DO LOOP runs faster than than any other type loop in any other language;

□ Multiple-nested DO LOOPS can all end on the one line – this is much more economical than the demands of all other languages;

□ Because FORTRAN is so precise and concise, programs must be written in blocks with copious comments included to be readable; and

□ Word length of variables, constants and function results can be chosen by the programmer.

The still-evolving FORTRAN language was first published to the world in 1956. Further developments added more capabilities, a revised enlarged version being the 1966 ANSI FORTRAN-66 specification published by the American National Standards Institute. After severe criticism regarding such shortcomings as the language's clumsy handling of files, text and alphanumeric characters, 1977 ANSI FORTRAN-77 was written – this is the version in use today.

To the credit of the designers, programs written under the rules of the early FORTRAN-4 will successfully Compile and run on machines equipped with a FORTRAN-77 Compiler. The same can be said for few other high-level computer languages.

FORTRAN for PCs

MICROSOFT FORTRAN-77 for IBM's PS/2 XT, AT, and PC, and of course the myriad clones, is equipped with the full repertoire of FORTRAN-77. Users of PCs running under the MS-DOS operating system can use the full capabilities of powerful FORTRAN-77 – all you need is two disk drives and at least 256 kilobytes of main memory.

Watered-down versions of FORTRAN have always been discouraged. Any implementations of reduced calculating power would reduce the reputation of this worthy language. Unfortunately, watered-down versions of FORTRAN exist – one that severely reduces the calculating power if the language is FORTRAN 77 for the IBM Personal Computer.

Most of today's FORTRAN Compilers are classed as 'smart' because they are able to change the program around somewhat to produce faster-running code and correct many errors. As an example of their smartness, these are a few of the error messages I've been presented with recently:

```
PREVIOUS WORD WAS MISSPELLED;
CORRECTED BY SYSTEM.

'END' COMMAND MISSING;
HAS BEEN INSERTED BY SYSTEM

SYSTEM CANNOT DECIDE IN WHAT
LANGUAGE PROGRAM IS WRITTEN
```

It usually takes about ten years for a high-level computer language to evolve from its conception to the publication of a formal definition. Small contradictions, imperfections or inabilities show up in a new language with use, particularly with unforeseen special cases of combinations. Theoretically, it seems that this cannot really be avoided, a theorem proved in 1931 by Kurt Godel states 'In any non-simple logical system, the axioms of that system cannot be shown in advance to be free of hidden contradictions.'

Recursion

THOUGH FORTRAN opened the eyes of the computer world to power and speed, and sold everyone on the the very idea of high-level languages, FORTRAN-4 does not allow re-entrant code or recursive program sections (this was a deliberate decision invoked to optimise run speed).

Recursive programs are those in which the same operation is repeated to simplify the total process to be performed. For example, the value of factorial 3 can be found by first finding 1 factorial, then 2 and finally 3 factorial. In other words, each recursion (reptition) is based on the results of the previous one. Re-entrant code is needed in programs in use by more than one user simultaneously – in large computer systems the Compiler, Editor and Linker programs may be used by perhaps ten or a hundred users simultaneously; the program needs to be able to be interrupted and then pick-up where it left off without losing any data from the various programs that may be in use at the time.

ALGOL

BY 1960 many programmers were wishing for a language which would allow recursiveness and re-entrancy. Because they also wanted subprograms that could be nested (a nested subprogram is a complete subprogram within another), an algorithmic language was invented – ALGOL-60 (ALGOrythmic Language-1960) – to allow nesting to the *nth* degree. Programs could be constructed with nests within nests within nests, almost ad infinitum. By its very nature, nesting is a recursive process, so fully recursive code and re-entrancy were allowed.

Many variations of ALGOL-60 appeared, including ALGOL-68, Burroughs Extended ALGOL, ALGOL-W and Euler; these widened the language's capabilities.

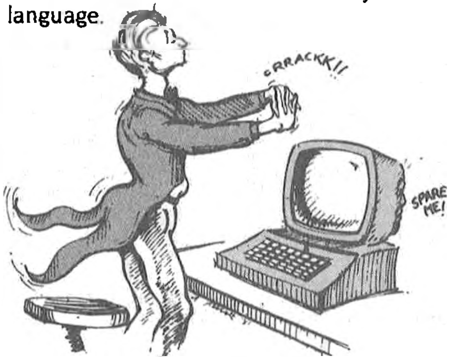
Around the same era, other programmers were thinking along different lines. Scientists and engineers had hitherto hogged the computer field with their long, involved calculations with small amounts of data. Now, others wanted to apply computers to recording business transactions and handling accountancy functions – this needed a different approach since these computations were quite short calculations on masses of data. This meant that quick, easy file handling was of paramount importance, along with character recognition and manipulation.

Today, though ALGOL is still used as a theoretical reference, little is written in that language. However, it became the parent of other languages such as PL/1, Euler, Pascal and Ada.

COBOL

COBOL (for COmmon Business Orientated Language) was developed in 1959 by a committee set-up by the US Department of Defense, which was interested in computerising its personnel records. A prime force on this committee was US Naval Captain Grace Hopper – in 1942 she coined the term 'bug': after several sleepless nights trying to make a seemingly perfect hard-wired program run, she discovered a burnt moth was short circuiting a critical pair of wires.

ANSI standardised this language in 1968 and issued a revised standard ANSI-COBOL in 1974. It is now a standard, machine-independent portable language used the world over. More programs have been written in COBOL than in any other language.



Business programming often involves more file reading and writing than other operations and the mathematical functions used are mostly the fundamental plus, minus, multiply, divide and percentage, with perhaps some statistics. Therefore, COBOL was designed to be strong in file work, capable in ordinary arithmetic but relatively weak in other more complex mathematical operations.

COBOL is one of the most English-like of computer languages, so much so that a program written in COBOL almost reads like an English language description of the problem to be solved. Thus, separate comment lines and documentation are not so necessary as with other languages.

All COBOL programs are divided into four divisions which can themselves be subdivided into named sections and/or other parts –

1) The Identification division identifies the program and the programmer, and perhaps includes the date written and where, states the program's purpose and whose files are used.

2) The Environment division is split into two sections, Configuration and Input/Output: the Configuration section describes the computer on which the program was written; and the Input/Output section relates each file mentioned to the tape or disk drive to be used, instructs the system to load the appropriate tape and disk drives and set up those particular input and output files.

3) The Data division is also divided into two parts: the File section describes the files, such details as record length and field description for each field in every file; then the Working/Storage section describes non-file data.

4) The Procedure division is subdivided into named paragraphs each of which is again subdivided into sentences – each sentence implements the actual calculations, functions and operations that the program is to perform.

COBOL's worldwide popularity was encouraged of its very English-like syntax; for example, 'Add 3 to page-number.' The powerful file handling capabilities are perhaps the major reason why COBOL is used today for many very large, wordy programs such as student record systems, library databases, business accounts and inventories, and government records.

The slow compilation time experienced on some computers with the language is not regarded as a great problem, for during the lifespan of a typical COBOL program most of its time is spent running; it's only compiled after modifications are made.

PL/1

IN THE early 1960s, while programmers appreciated the mathematical capabilities of FORTRAN, they bemoaned its uneasy file handling. Similarly, some loved COBOL's great file handling power, but thought the mathematics inadequate for their purpose. Furthermore, they complained that in wordy COBOL you 'Don't so much write a program, you write a book!' ALGOL, the critics said, needed more power in file work.

IBM replied to all the above criticism by designing PL/1, endowing it with most of the good points of FORTRAN, COBOL and ALGOL. PL/1 (for Programming Language 1) was published by the IBM laboratories in 1964. For some reason, PL/I is often referred to as PL/I – a typing error which stuck, perhaps?

Full PL/I possesses a vast array of capabilities, has no reserved words, is good at calculation ***and*** good at file handling. But now, critics complained of the size of the

...NO WORRIES OL'MATE IF YA WANT MY ADVICE GENUINE BARGAIN OF THE WE
..NO SWEAT LIKE A CHARM SOMEONE ELSE'LL GRAB IT DO YOURSELF A FAVOUR
...IT CAN'T LAST AT THIS PRICE RIPPER DEAL SUIT NEW MICRO BUYER HA
..HUGE DEMAND GOT ONE AT HOME JUST LIKE IT OTHERWISE I'D BUY IT MY
HONEST ELTON USED MICROS
LOG BOOKS LOW K'S GOOD LADY'S MODEL
THIS WEEK'S HOTTY!... THRIVES ON SPACE INVADERS!
GENUINE ONE OWNER... LIGHT TOUCH PROVERBIAL LITTLE OLD
ASK ABOUT A TEST RUN.
LOW DEPOSIT EASY R.O.M.
ONLY 70,000 CHARACTERS ON THE CLOCK
A.C.S. INSPECTED!
©B.J. AKHURST. 82

So You Want To Buy A Second Hand Micro?

language and the difficulty of learning it. Furtheremore it requires a large compiler which won't even fit in some mini-computers, let alone small micros!

The inevitable happened: a watered-down version called PL/C was produced using a smaller compiler. PL/C contains 40 reserved words and has many fewer features than PL/1.

The dropping of these PL/1 features tends to make a PL/C program longer – so much so that there generally isn't much advantage in using it.

PL/1 gives the programmer valuable options in basic arithmetic functions. For example, they can specify the number of decimal or binary digits the arithmetical operation is to use, and (in fixed point format) how many digits will follow the point. In other languages this can be a complex and time-consumng procedure.

Properties of PL/1

IN ADDITION to those properties of full PL/1 noted in the text, other characteristics of the language include –

□ No reserved words;

□ the language is more free-form than FORTRAN, less English-like than COBOL and has many similarities to ALGOL (abd Pascal and Ada);

□ Each program statement is terminated by a semicolon;

□ PL/1 is not column conscious – you can write anywhere on the line;

□ the declaration of data and variable types is more clumsy than in FORTRAN;

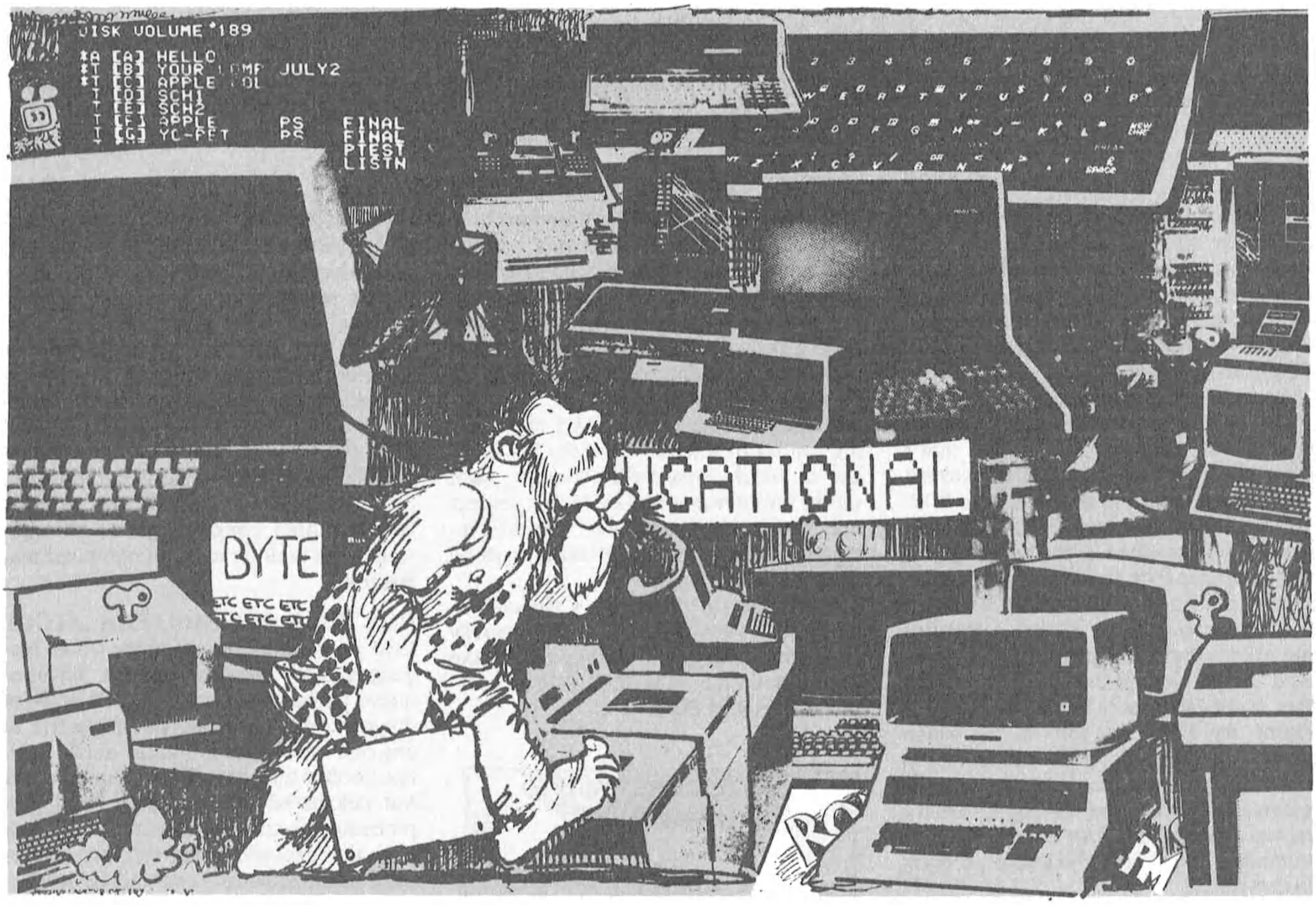
□ Recursion is allowed; and

□ Nesting is allowed.

As you can see, PL/1 is an excellent language, capable of fully controlling every aspect of the running program. It is endowed with a very powerful set of commands, expertly put together by those who have suffered the inadequacies of FORTRAN, ALGOL and COBOL.

Part 2

IN PART 2 we'll begin with a comparison of the block structures of various languages – block structures (and the three philosophies behind their use) greatly affect the method of programming and have a strong effect on the time it takes a program to run. Then, we'll cover the BASIC (Beginner's All-purpose Symbolic Instruction Code) family of languages, often called the poor man's FORTRAN. □



Start Computing - Part 2

In Part 1 we started computing with an overview of the development of computer languages – let's continue with 'block philosophy' and 'the poor man's FORTRAN', then a brief look at Pascal.

THE BLOCK Structure of any language dictates the programmer's attitude and methods, so let's consider the Block Structures of various languages. FORTRAN, ALGOL and PL/I can all be divided into separate blocks, but the rules concerning blocks are different in all three languages. So different, in fact, it's difficult to think of the blocks in the three languages as being similar. So, for the time being let's just think of a block as a 'building block' – the smallest unit of a program that produces a result (just as a program is the result of putting hundreds of blocks together, so too, the final 'answer' is the result of putting together the 'answers' from those hundreds of blocks).

We'll start with a look at how each of the three languages uses blocks – and then derive a philosophy of blocks – this philosophy has a great effect on the speed with which a program will run. Much of the following may seem extremely esoteric, but a basic understanding of language blocks is necessary to understand why one language is 'faster' than another, or why a particular language is used for a particular application.

FORTRAN blocks

IN FORTRAN a block is either the compulsory Main program, or a subprogram; subprograms are either subroutines or function subprograms. Any subprogram, or group of subprograms (with or without the Main) can be written, error checked (debugged) and compiled separately. All subprograms are then linked, together with the Main, prior to running.

FORTRAN blocks are not nested within each other, rather all are separate worlds, each with its own environment of declarations of type and dimension for each variable. It may not be immediately apparent, but this is the keystone of every large

Start Computing was written by Bryan Maher, a freelance technical writer and formerly senior lecturer in computer science at the Capricornia Institute of Technology, Townsville Qld.

FORTRAN program and sets the language apart from all others. This property is an enormous asset, greatly simplifying the job of writing, compiling and debugging long programs. It is also instrumental in increasing the running speed of many FORTRAN programs.

Because of the discrete nature of FORTRAN subprograms, very large programs can be written piecemeal (in fact, that's often the only way they *can* be written). The usual approach in writing a large FORTRAN program is to write the Main program and the easiest of the subprograms first; these are then linked and set running (often producing useful results), while further subprograms for extended features are written and debugged ready for inclusion at the next linking session (during this stage, references (calls) to subprograms are preceded with a 'C' which makes the compiler think the line is a comment, so it's ignored). As the subprograms are finished, the dummy Cs are removed – this process can be repeated any number of times until the whole program is complete.

Since FORTRAN blocks are independent, no block knows any of the variable types, dimensions or values of another block unless its told. Also, recursion is not allowed in FORTRAN and no block can call itself, although any block can call any other subprogram any number of times. Subprograms can be written, compiled and linked in any order, but at run time the computer will seek out the Main Block, where execution of the Run program will begin.

PL/1 blocks

PL/1 CAN be divided into blocks, of which there are two types – Procedure and Begin blocks. There must be at least one Procedure block, but then there can be any number and one must be labelled Main.

In contrast to FORTRAN, PL/1 Language Procedure Blocks may be written within some other Procedure, or written externally (separately), but everything must be contained within the Main Procedure.

In PL/1, the program enters a Procedure block only by a Call statement or a Function reference. In this respect, entry into a PL/1 Procedure appears somewhat like entry into FORTRAN Subroutines or Function Subprograms. Then again, entry into a PL/1 Begin Block is not unlike entry into FORTRAN DO or IF Loops. However, environment considerations (and hence speed of execution) are quite different in the two languages.

In FORTRAN each Subprogram is a complete isolated environment which fully includes all Loops within it. No Subprogram is nested inside any other Subprogram. Contrast PL/1 where Procedures may be nested within other Procedures. In such a case a variable Declaration in an outer Procedure is known to all other Procedures nested within. And, this nesting can go to unlimited depth, any number of Blocks may be be nested, each within the previous Blocks. In this respect PL/1 is very, very ALGOL-like. The only restriction is that each nested Block must be contained completely.

Many users believe PL/1's declaration rules lead to endless confusion, especially in very long programs, so they opt for the simple and clear FORTRAN rule of completely isolated Blocks.

Block philosophy

IIN LONG programs, or even in short programs with involved iterations, fast Run time is of paramount importance. Although not immediately apparent, a language's Block rules have great bearing on the time the program will take to Run. Both the method of storage environment used by a language, and the rules for calling Subprogram Blocks in each language affect running time of the program.

Computer languages can be divided into three groups according the block rules and type of memory storage environment each uses –

BASIC, COBOL and ***FOCAL*** use a single large storage environment for the whole program, including all subprograms. In BASIC, any command can access any variable no matter where in the program it occurs. For example you can GOTO any line, thus every line must have a different line number.

BASIC programs run slowly for a number of reasons. BASIC's interpretive nature (which we will examine in detail later in this series) is a prime cause of poor running speed. But also to blame is the method the system uses to find other parts of the program referred to – for example, if a BASIC program is at line number 2412, and meets a command GOSUB 2276, the program needs to return to line 1 and then trace down through thousands of line numbers looking for the required line. It cannot jump straight to the subroutine as is done in other compiled languages.

PL/1, Pascal, LISP, APL, Euler, ALGOL (and its variations) and many other languages, use separate storage environments for each subprogram. Only some of these storage environments are active at any one time. Thus in Pascal, actual storage needed for a procedure subprogram is not calculated and allocated until that procedure is called and entered. On exit from that procedure, the memory storage used is de-activated, and can be re-allocated by the CPU to any other use. So it is possible to fit a large program, broken into small procedures, onto a small computer.

In PL/1, with its multiply-nested Procedures, one procedure can call another nested at the first or second level below. However, no procedure can call another nested deep within itself, if it is nested more than two levels down. This is because only some procedure storage environments are active (that is, memory addresses allocated and known to the CPU) at any one time.

One reason for slow running time of this group is that each time a procedure is called, the CPU has to stop what it is doing and set-up the memory storage needed and the relevant environment declarations. Repeated Calls to a procedure therefore incur much time lost in repeated allocation and setting-up of memory storage and the environment required.

If a procedure calls another nested within it, all declarations in the outer procedure are still active and form part of the inner environment. But if the system comes to a new declaration of a variable already in use while halfway through the inner procedure, again it must stop its computations until it has set-up more memory address space for this different version of the variable, still keeping the old one declared in the outer.

Each time ALGOL calls subprograms, all variables passed to the called proce-

Grandchild of FORTRAN

WHILE THERE are many versions of BASIC, they can all be considered descendants of the original FORTRAN. Those characteristics inherited from that language include –

1) Both languages are line-conscious, meaning normally you type one statement per line and pressing the Return key terminates that statement. No, end-of-statement punctuation is necessary, in contrast to ALGOL-like languages such as Pascal.

2) When you want to write an equal sign in BASIC, the ordinary = you have used since childhood will do, (none of this silly := or even ::= as demanded by ALGOL-like languages.

3) In both languages the powerful (but dangerous) GOTO and conditional GOTO statements are allowed, and the GOTO can be either in forward or backward direction.

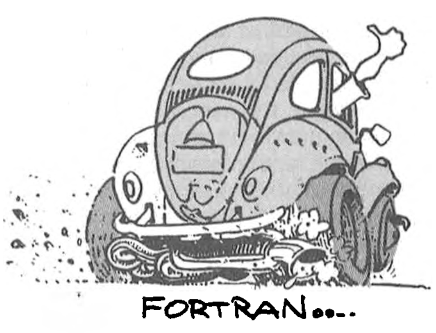
4) The finished BASIC program, like one written in FORTRAN, is clean, concise and easy to read (provided copious comments are included). It is not cluttered up and riddled with the excesses of over-done punctuation that plague ALGOL-like languages.

But then of course there are differences between BASIC and grandfather FORTRAN, some arising from the original simplification step-down, such as the dropping of complex numbers and complex functions. In the other direction one striking difference is that the BASIC language has reserved words, in fact, 'standard' BASIC has 184 of them (nearly as bad as COBOL)!

dure are 'passed by name' only. As explained below, this can result in much time consumed in CPU calculations to establish the actual value of the passed variable.

FORTRAN also allocates separate environments for each subroutine and function, but only in FORTRAN are all environments established at the start of execution. When a FORTRAN program is executed, the memory necessary for each subprogram has already been allocated during compiling and linking. Therefore all environments are active all the time the program is being executed, all memory requirements for all subprograms are allocated, and all variable types and dimensions are known to the CPU.

This philosophy results in more memory being tied-up the whole time (compared to execution under other languages), but, immediately any subprogram is called, the CPU can get straight on with performing the necessary calculations in that subprogram.



A further important aspect of the FORTRAN philosophy is that in a call to a subprogram all passed variables are 'passed by reference', meaning that the memory addresses of the passed variables are passed to the called subprogram. Speed of program execution of FORTRAN programs is enhanced by this method of 'calling by reference' because at the moment of a call, no time is lost in calculating the values of variables to be passed.

Contrast other languages, such as ALGOL, where a call only transmits the name of the passed variable – the CPU needs to stop what it is doing and calculate the variable each time it is called.

The BASIC family

STANDING FOR Beginner's All-Purpose Symbolic Instruction Code, the BASIC Language was written in 1965 by John Kemeny and Thomas Kurtz at Dartmouth College. It was meant to be a simple language in which beginners could quickly learn to write and run programs, so they designed the language to be interpretive (rather than compiled) to save students the task of compiling and linking.

They succeeded in their aim admirably – the personal computer's success depended heavily on the popularity of BASIC. Sometimes it is called the 'poor man's FORTRAN', but it might better be described as a very large, loosely related family of variations, all members being 'grandchildren' of the original FORTRAN.

Many different versions of BASIC have appeared. The IBM-PC on its own supports no less than four (essentially) different versions: ROM-BASIC, Disk-BASIC, BASICA and Compiled BASIC. Many mini-computers, such as the VAX 750 and 780, HP's machines and many others, there are different versions of BASIC again. BASIC has been so altered, simplified, extended, updated, re-hashed or adulterated over the years, that it's often necessary to re-write a program to run on a particular machine – no wonder it's called a 'family' rather than a 'defined language'.

Reserved words

RESERVED words are those whose meanings are defined within the language. They must be used in the right place (they cannot be used anywhere else, especially as variable names or arguments).

Let's consider the reserved word AUTO – it's used if you want the BASIC editor to automatically increment program line numbers as they are typed in. While you shouldn't use AUTO as a variable or as a variable argument, some BASIC editors will let you break this rule – then all sorts of things will go wrong when you try to run the program. The worst thing that can go wrong is for the program to run, apparently quite happily, but giving a rubbish to six decimal places! (At the other extreme, the BASIC editor on a VAX-11/750, will automatically delete a program line if a reserved word is misused.

However, AUTO can be used if it is buried within some other word, without any delimiters: AUTOMATIC, for example, is acceptable, but the hyphen in AUTO-MATIC may be seen by some BASIC interpreters as two separate words, one of them the reserved word.

Contrast this with FORTRAN which prides itself as being one of that very rare breed of computer languages that have no reserved words at all.

BASIC potency!

TO THE credit of BASIC, for a 'little' language (at least in some implementations), it is extremely versatile, wide ranging and very potent. We might say 'powerful' too, except many computer buffs and much advertising misuses that word to mean computer speed – in no way can we ever call BASIC (nor any interpretive language) a fast language.

Its potency and versatility come largely from the very fact that BASIC is interpretive, implying that the source program (the program you write) comes into intimate contact with the computer's operating system line-by-line. Indeed BASIC enjoys the privilege of being able to reach right into the operating system, a technique denied to FORTRAN except in cases of a specially written operating system.

In terms of the extras inserted into the BASIC language, it seems the grandchildren grew and, at least in some directions, became more 'powerful' than the grandfather. Some of these 'extras' seem simple enough, until you try to perform the relevant function in some other language – take the BASIC command CLS, which simply clears the screen.

So You Want To Write Instructional Programs?

It is so commonly used from within BASIC programs that we take it for granted. But CLS is actually an operating system command – you can type CLS anytime you are in the environment of most operating systems and it works. Whether your screen is blanked by the slow process of writing a blank character to every screen address, or by quickly resetting the whole screen register on the display driver card, is a detail BASIC doesn't have to worry about – the obliging operating system does the work.

Perhaps the most useful aspect of BASIC's ability to reach right into the operating system lies in the language's graphics commands. Particularly strong are those found in BASICA and its derivatives. For example the CIRCLE command will draw circles or ellipses all over the screen, DRAW can fill your screen with straight lines and PAINT can be used to colour in the resulting shapes. This is not all idle fun – the drawing may be graphs of some vital process.

The graphics commands are quite different from text commands as graphics commands can reference any screen locations, in any order, to commence or continue drawing shapes. Text commands, however, must always begin at the location following the current screen location.

If we use cars as an analogue for languages then BASIC is the family sedan – it's inexpensive and comes with all sorts of optional extras

An interpretive nature

BASIC's interpretive nature makes it very different from all compiled languages. To use BASIC, you first write the program in the pseudo-English of the language commands, then you can immediately run it. At the RUN command your program is fed, line by line, to the BASIC interpreter (a permanent program resident either in a Read-Only-Memory (ROM) integrated circuit or in a corner of the computer's main memory).

The interpreter takes each program line and converts it into a segment of machine language instructions (interprets it), feeds that machine language segment to the CPU and the results are stored for later use.

As an example, consider this little section of a program (EXP is the reserved word for the natural logarithm, *e*) –

```
 50
 60
 70 Z=4
 80 PRINT EXP(Z+2)
 90
100
```

When the program is running and comes to line 70, the value 4 is stored in the memory address allotted by the program to the variable Z. Then the program comes to line 80 and the interpreter tackles the conversion of that line into a machine language segment. Starting inside the brackets, first the value of Z (4) is fetched from memory, then the number 2 is added giving the whole bracket the value 6. Then a truncated infinite series is set-up in machine language to calculate the truncated approximation of e^6.

The usual series used by interpreters would be –

```
e 6 = 6 0   6 1   6 2
      --- + --- + --- +
       1     1    2*1

         6^3 ... to 28th term
         ---
         3*2
```

where the top line is a succession of rising powers of 6. In the the bottom line, each term is the value of the previous bottom line term multiplied by whatever number is used as the power above. The mathematicians amongst us would call that top line a power sequence, and the bottom line would be called a sequence of factorials.

In principle, this series is infinitely long, but computers must stop the calculation somewhere, and truncating (chopping off the infinite tail) the series after the 28th term does introduce some error. But, after adding up 28 terms the program line has the value 403.428793 and the 29th term would only add another 0.000 000 004168. So execution of the line is completed by rounding to 403.4287 and printing that number on the screen.

Then – all the machine code generated to execute line 80 and its result is thrown away, after being used only once!

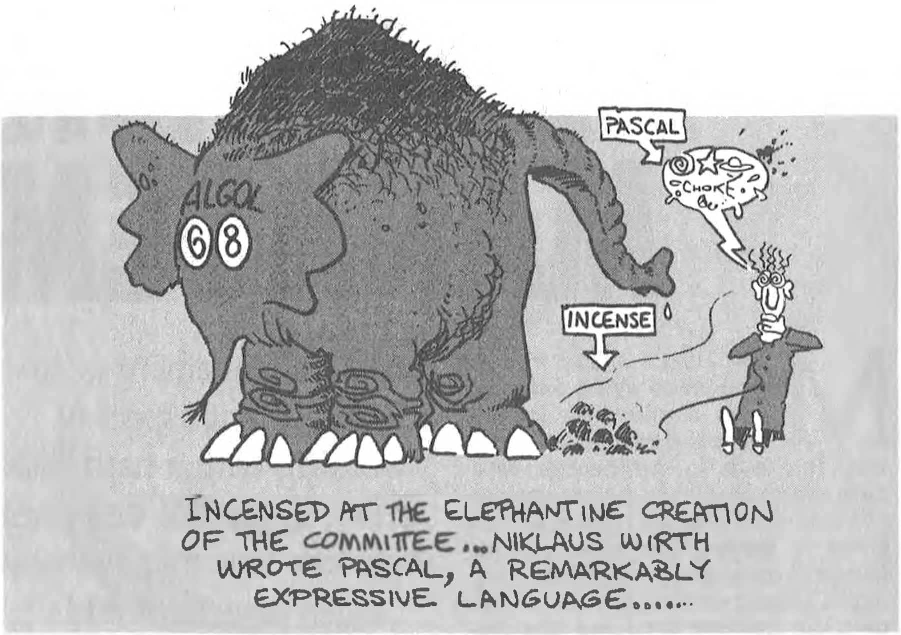
Should line 80 happen to be within a loop, probably with the value of Z changing at each pass, that whole lengthy process of setting-up all the machine code for that line has to be done over and over again at each pass through the loop. Now you see why interpretive language programs *must* be slow running.

Contrast this repeated setting-up and throwing away of machine code with the compiled language's method: the compiler sets-up machine code similar to the above description, but the vital difference is that the compiler sets up all the machine code once only at the compile stage. This is kept as the object file, to be linked to make an executable file which can be run as often as you wish with having to perform all those time-consuming calculations.

There's another factor that makes interpreted programs slow – during interpretation, any GOTO or GOSUB commands cause the computer to trace through the full program listing, starting at the first line, until it comes to the line number referred to. If a subroutine is located near the end of a large program, and if the call is within a repeated loop, this searching can consume so much time you wonder if the machine is asleep.

Similar delays occur every time the program has to access any of the variables, as they have to be found by searching since no absolute addresses of memory locations of the variables are known to the BASIC interpreter or to the operating system.

Contrast compiled languages, where the run file knows the absolute (or relative) addresses of the targets of all GOTOs and subprogram calls, allowing direct access to the required branch of the program. Furthermore the run file knows exactly where each variable is stored in memory at run time, so variables are found quickly and directly.


ALGOL
68
PASCAL
CHOKE
INCENSE
INCENSED AT THE ELEPHANTINE CREATION OF THE COMMITTEE...NIKLAUS WIRTH WROTE PASCAL, A REMARKABLY EXPRESSIVE LANGUAGE......

Pascal

NIKLAUS Wirth, a Swiss computer scientist had done much development work on ALGOL, one of the favorite computer languages of the 1960s. He was also quite conversant with the four other popular high-level languages: FORTRAN, COBOL, ALGOL and PL/1 (an overview of these was given in Part 1). He took his own improved ALGOL a large step further in 1971 and published his own language, with all the characteristics he considered desirable.

He called it Pascal, to honour Blaise Pascal, French mathematician (1623-1662). Wirth is said to have invented his new language because he tired of teaching program construction which could not be explained logically. So, Pascal forces students to write organised, structured programs.

Largely following the earlier ALGOL (from which Pascal inherited the word 'procedure' and ALGOL's excessive punctuation rules), Wirth went further and endowed Pascal with more data types than any previous computer language. From COBOL he took the requirement that the program and its input and output files be named. From FORTRAN he used the idea of optional comments. From ALGOL and PL/1 he was encouraged to endow his new language with recursive power.

COBOL

From COBOL the idea of simple English-like commands IF-THEN-ELSE, DO-WHILE and REPEAT-UNTIL enamored this new language to students. He hated spaghetti-code; the positively unreadable lines of a high-level program that can result when long programs are written without sufficient thought, discipline, blocks or comments. Therefore, he incorporated strict rules of structure in his new language.

From FORTRAN he endowed it with 'calls by reference', but he also added also 'calls by value' after the fashion of ALGOL.

Pascal is not generally regarded as a very mathematical language; many implementations do not support complex numbers, nor the double-star (**) notation for powers and roots. So, the inevitable had to happen: enhanced versions of Pascal appeared on the scene. In defence of Pascal, it must be said that the language is quite good at file handling, is a good choice of language for 'pattern-generation' programs, such as those generating printed circuit board diagrams and layouts for integrated circuit microchips.

And that completes our overview of computer languages. Of course there are many more than we've discussed here, but, by now you should have an understanding of why there us such a plethora of languages and the type of application each is best-suited for. In Part 3, we'll come back down to earth with a look at 'Computer Software – Simply!' □

HARD DISK FILE MANAGEMENT

File management is an important aspect of working with a hard disk drive. Bruce Iliff discusses how effective management can save time and confusion.

MANY USERS simply load up their drives with a congealed mass of files then stumble along keeping the system running. This leads to inefficiencies, especially if many people use the one machine.

The directory structure is the main component in keeping files in a logical sequence. In many hard disks the root directory is clogged with a multitude of files, the most common being the DOS files. These are usually copied from disk when setting up the system and are then just left in the root directory. Other files that have no logical place are added and the root directory becomes a conglomeration of files, making it difficult to get any sense from a listing.

To overcome this, the root directory should contain only the three important system files: command.com, config.sys and autoexec.bat. This leaves the root directory as a listing of the main sub-directories.

The directory tree

KEEP THE directory tree wide and shallow with a maximum of three levels. A word processing directory might be –

```
C:>\word\letters
```

This ensures any searches don't have to scan through a multitude of branches to find a particular file and cuts down on long path names. It's also important that, where possible, each application has its own sub-directory and the data files are kept in a separate sub-directory. This makes it much easier to find which files go with what program when it's time to update the application or delete it. With that arrangement, deleting wanted data files or program files when you want to delete the other is much less likely to happen.

Use many small directories instead of one filled with numerous files. Don't put all data files in a sub-directory Data under the dBase directory. Instead group them into their particular category. For example –

```
C:>\dBase\Proj1
```

would be for all files about Proj1 and another sub-directory for Proj2. This makes it easier to see what is in each sub-directory and cuts down the time to locate a file. There's no reason that text or other files relating to those projects can be kept in the same sub-directory.

Use directory names that make sense. A directory containing Letters to Foreign Companies could be LTFC, but a more logical name would be just Foreign. The period separator can be used in directory names. The Foreign Letters directory

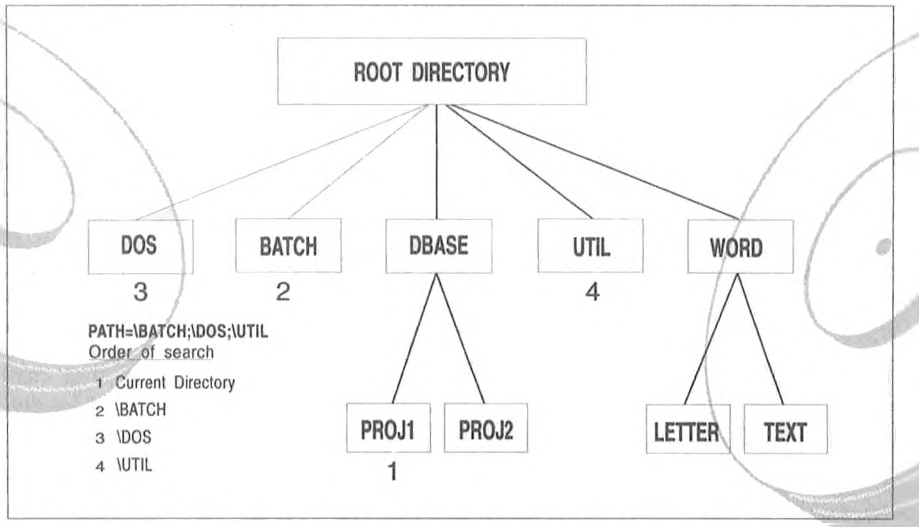
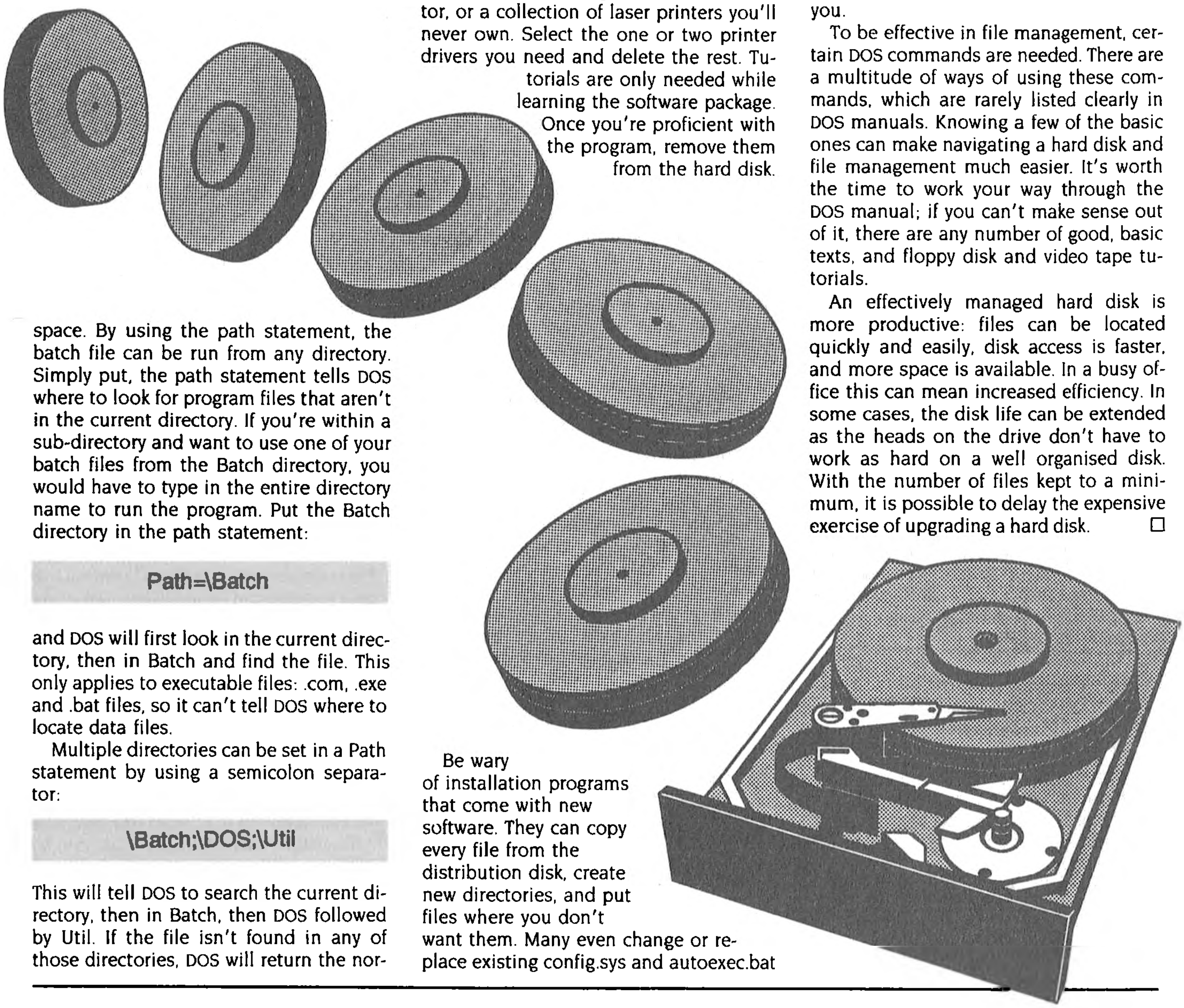


Figure 1. One tip for efficient use of a hard disk is to make good use of the path statement in autoexec.bat.

could be distinguished from Foreign Data by using Foreign.let for letters and Foreign.dat for data.

Create separate directories under the root directory for batch files, the DOS files and any utilities. This makes the task of locating them much easier and guards against accidental deletion. If old program files are left scattered over the disk, some could be forgotten and play havoc with the new version. Utility programs might be useful packages like XTPro or PC Tools. Having these files in one directory means they can be run from anywhere on the disk with the path statement.

Many users of DOS are not aware of the power of the path statement. Some inexperienced users who want to run a batch file from anywhere on the disk, copy the file to each sub-directory wasting valuable space. By using the path statement, the batch file can be run from any directory. Simply put, the path statement tells DOS where to look for program files that aren't in the current directory. If you're within a sub-directory and want to use one of your batch files from the Batch directory, you would have to type in the entire directory name to run the program. Put the Batch directory in the path statement:



```
Path=\Batch
```

and DOS will first look in the current directory, then in Batch and find the file. This only applies to executable files: .com, .exe and .bat files, so it can't tell DOS where to locate data files.

Multiple directories can be set in a Path statement by using a semicolon separator:

```
\Batch;\DOS;\Util
```

This will tell DOS to search the current directory, then in Batch, then DOS followed by Util. If the file isn't found in any of those directories, DOS will return the normal 'Bad command or file name'. Don't set the path to look at a floppy drive, this is asking for total confusion!

The most dangerous file on any hard disk is format.com. With a few misguided keystrokes, this can format the hard disk. Later DOS versions require specific keystrokes to achieve this, but it is a dangerous file to have around in its basic form. As it is also needed to format floppies, call it Dont_use.com and write a batch file called format.bat with the line Dont_use A:. Then the command Format will format the floppy in A: drive.

Installing software

WHEN INSTALLING new software, only copy the files you need for your application. There's no need for drivers for an EGA screen if you've got a Hercules monitor, or a collection of laser printers you'll never own. Select the one or two printer drivers you need and delete the rest. Tutorials are only needed while learning the software package. Once you're proficient with the program, remove them from the hard disk.

Be wary of installation programs that come with new software. They can copy every file from the distribution disk, create new directories, and put files where you don't want them. Many even change or replace existing config.sys and autoexec.bat files. Before running an installation program, make copies of these files with a .bak extension.

Regularly go through your files and remove unwanted ones. A lot of software creates duplicate files when the program saves to disk, giving them a .bak or other easily identifiable extension. This can quickly lead to a vast collection of files.

Any files you haven't used for a long time, archive onto a floppy disk. They might be accounts from the previous year that aren't required for daily use, but need to be kept. A floppy disk in a safe location is the best place for this type of data, not wasting space on a hard disk. Don't store your back-ups in the same building as the computer if at all possible – an easy way to do that is to put the back-up in your briefcase or jacket pocket and carry it with you.

To be effective in file management, certain DOS commands are needed. There are a multitude of ways of using these commands, which are rarely listed clearly in DOS manuals. Knowing a few of the basic ones can make navigating a hard disk and file management much easier. It's worth the time to work your way through the DOS manual; if you can't make sense out of it, there are any number of good, basic texts, and floppy disk and video tape tutorials.

An effectively managed hard disk is more productive: files can be located quickly and easily, disk access is faster, and more space is available. In a busy office this can mean increased efficiency. In some cases, the disk life can be extended as the heads on the drive don't have to work as hard on a well organised disk. With the number of files kept to a minimum, it is possible to delay the expensive exercise of upgrading a hard disk. □

Spee

UP A HARD DISK

Here's advice on efficient use of a hard disk from Bruce Iliff – it could save you replacing that old clunker.

THE COMPUTER INDUSTRY is becoming obsessed with speed. Fast computers that were once a dream in a scientist's mind are now commonplace in the home. In this quest for faster machines, many users pour money into hardware and neglect the small changes that can increase the speed and productivity of their computer.

The computer's speed depends mainly on the processor. An 80386 chip is faster than the 8088 and the new 80486 is the fastest of them all. The only way to increase this component of a PC is by upgrading. A maths coprocessor can speed particular applications that require a lot of mathematics, like CAD packages. The other component that has a direct bearing on speed is the hard disk. It's no good having a super-fast 80386 coupled with a slow disk, especially if an application requires continual disk access.

Unlike the main processor, it is possible to get better performance without expensive upgrading. Money can buy speed, but bad disk management can quickly slow it down. A well managed slow drive can be quicker than a technically faster one. There are a number of ways to get this better performance through simple software procedures.

Fragmentation

WHEN DOS writes to a disk, it has a certain order it uses to find available space. On a newly formatted disk a file will be in contiguous sectors. If the file increases in size, part might stay in the original location and the rest put on another part of the disk. As files get erased and other files added, this process keeps growing until files are split into pieces and scattered around the disk. The files are now fragmented.

On a badly fragmented disk, the head will have more movements to locate all parts of a file. It might start on the outside edge of the disk, the next portion might be on the inner-most section, and the rest near the outside. This not only decreases the speed, it can also affect the life of the drive as the heads are having to do more work.

The first thing to do if you suspect your disk could be suffering from fragmentation is to check it by running chkdsk on a particular file, for example –

```
CHKDSK \WP50\WP.EXE
```

This command will tell if the file wp.exe is contiguous. Try it on a few large files and if they are all fragmented the disk could probably benefit from de-fragmentation. There are a number of ways of doing this. The cheapest is to make a global back-up, reformat the hard disk then load the files back on. Make sure you take two global back-ups in case something happens during the process.

This is a slow and painful way. A quicker method is to use a defragmenting program. Popular ones are the defragmentators contained in Norton Utilities or Mace Utilities (both of which are available from any good dealer). They shuffle files around on a disk, piecing them together then locating them in contiguous sectors. If running one of these, it is wise to take a back-up of important files as some problems could occur, like power failures while the defragmentation is in progress. Depending on the size of the disk, this process could easily take over an hour. These two software packages have many other uses for hard disk management.

The amount of time between de-fragmentation depends on how much file erasing and file manipulation goes on. Once a month would be a good time frame for most hard disks. Including a global back-up at the same time would be a sensible management schedule.

RAM disk

THE USE OF a RAM disk can help speed an application. A RAM disk, or virtual disk, is part of the system's main RAM set aside to act as a disk drive. Files that require continual use can be put in the RAM disk and accessed almost instantaneously.

There are many applications for this. In most of the major wordprocessors, not all the code gets loaded into memory. Extra files remain on the disk and continual access to these can be annoying. Sometimes this can damage the disk surface as the head makes continual reads of the one section of the disk.

Another application is the text editor in a database program. If using dBase III+, try setting 'modi comm' to read the shareware program Qedit, or your favourite editor, in a RAM disk. This can increase productivity and do away with that long wait for the editor to load from the hard disk.

To make the RAM disk put the line 'device=vdisk.sys' in the config.sys file and reboot the machine. This will create a RAM disk with 64Kb of memory, 128 bytes to a sector and 64 directory entries. Other settings can be invoked depending on your application. Using expanded or extended memory is perfect for a RAM disk.

With a single hard drive called C: the RAM disk will be D: drive. Put a statement in autoexec.bat to copy the required files to the new disk. Also check path statements to ensure DOS knows to look in the RAM disk. And be sure the word processor or database program knows about the RAM disk, or it could still access the hard drive.

The RAM drive works exactly like a normal disk drive, except when the power goes off! As RAM is volatile memory, all the files in the RAM disk will be lost. If using the RAM disk for data files, be sure to save them to a disk before switching off.

It's no good having a super-fast 80386 coupled with a slow disk.

Interleave

SECTORS ON A hard disk are in tracks, with 17 sectors to a track on a typical DOS machine with each sector holding 512 bytes. The drive head reads the entire 512 bytes of the sector as the platter moves under it. Then the data gets passed to the main processor, and the head prepares to read the next sector. Sometimes the head might miss the next sector when it passes underneath so has to wait for the disk to go around one more time, slowing the access time. The reverse works when writing to the disk.

To get over this problem, the 'interleave' is adjusted so the sectors are not together but have another sector between them – see Figure 1.

Buffers and caching

DOS BUFFERS contain the latest sectors read from the disk. For sectors that are continually read, the data gets stored in a buffer and is read from there rather than the disk. This works in reverse when writing to the disk. In a data file, if a 256 byte record gets changed the data gets held in a buffer. Then, if the next 256 byte record is also changed, the buffer gets written to the particular sector. This writes both records in one disk operation instead of two. The number of buffers has to be set in config.sys. DOS can cope with a large number of buffers, but care should be exercised as each buffer is 528 bytes, so too many can waste precious memory. Time can be wasted searching through all these buffers and still result in a disk access in the end. About twenty buffers is a good average.

When an application program uses buffers, it occasionally flushes them to disk if they haven't been accessed for a time. Buffers are also written when exiting the application – one reason why it is important to leave applications through their proper exit procedures.

Disk caching is similar to the use of buffers, but can hold entire files in memory. It can be considered a combination of a RAM disk and buffers and is best used with extended or expanded memory.

A carefully planned directory structure can decrease hard disk access time. If wanting a file deep within a directory tree, DOS has to search through all the sub-directories in between until it comes to the file. In the same way, care should be taken when using the 'path' statement.

Set up the path so the first directory searched is most likely to contain the correct file. If you are using a lot of batch files in the directory Batch and occasionally need a file in the DOS directory, and only at the end of a session need to access the Menu directory, set the path to read the directories in order of preference –

```
PATH=\BATCH;\DOS;\MENU
```

Unwanted files are another trap that slow disk drives. In a directory clogged with 100 files, the heads have to go over about 50 files on average every time it accesses the directory. Include this task of deleting unwanted files in the global backup and de-fragmenting process.

Fastopen, a DOS command in versions 3.3 and greater, sets aside a portion of memory that holds the address of recently accessed files. When a file gets accessed again, DOS knows exactly where it is on the disk and the heads can go straight to it.

By using a combination of these methods, it is possible to decrease the access time of a hard disk. All these techniques require constant thought and planning. There is little logic in de-fragmenting a disk and then forgetting about it. As fragmentation is a continual process, it must be done regularly. A suitable RAM disk can cut down on disk accesses only if set up right, and buffers can actually slow access time.

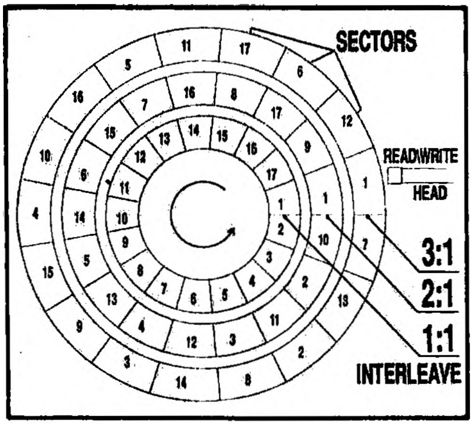


***Figure 1.** With a 1:1 interleave on a hard disk, the sector numbering would be 1, 2, 3, 4, 5 to 17. With a 2:1 interleave, the numbering would be 1, 10, 2, 11, 3, 12, 4, 13 up to 17. In this arrangement the head would read sector 1, pass the data on while sector 10 moves underneath, then it is ready when sector 2 passes underneath. A 1:1 interleave gives the best performance, but it also gives the greatest penalty if not correct as the disk has to go around one more revolution if it misses the sector. With a 3:1 interleave, the head is sitting waiting for the disk to move only a fraction before it can read the sector. To change the interleave requires formatting the disk.*

Continual reviews of your system are required to ensure it is in peak operating order. Just like a car requires regular servicing, a hard disk requires regular 'software' servicing. As many of these tasks should be done regularly, group them together to ensure they are done. Once a month is a good average time. First check for unwanted files, do a global back-up, de-fragment the disk, and check the RAM disk has the correct software in it for the current usage of the machine. Then the hard disk should be ready for another month of heavy use.

It is surprising what gains in speed can be achieved with a little thought. By applying these techniques, it is possible to delay expensive equipment up-grading. □

EXTENDED, EXPANDED, ENHANCED EMS, EXPANSION MEMORY

WHAT DOES IT ALL MEAN?

Are you still staggering along with that old PC or XT, regularly dreaming of breaking the 640K barrier? Plug-in memory expansion is the obvious answer – if you can decipher the semantic confusion surrounding memory cards. Stewart Fist translates . . .

WHAT IS THE difference between extended and expanded memory? And which of these (if either) does 'memory extension' refer to? And why would you use one technique and not the other? Can both be used together? And where does EMS and Enhanced EMS come into all this?

You aren't alone in your confusion. A few months ago I was researching an article on accelerator cards and I talked to technicians from a number of card suppliers. Half of them didn't have a clue about the reason behind the PC's original 640K limit, or how extended and expanded memory differed – and the other half thought they knew, but had it wrong.

So let's get hard-nosed about this, and dispel some of the myths.

First of all, MS-Dos doesn't have a 640K limit to the memory size – this myth results from an architectural decision made by IBM when they designed their first 8088-based PC. It carried on in the 8086-based XT range, and then into the 80286 and '386 machines when they are running in 'real' mode. IBM's reason was simple: they wanted to have a firm, fixed location for their video RAM and, as the streaker said to the magistrate, 'It seemed like a good idea at the time' to place this in the address space starting at 640K.

If you want proof that this wasn't Microsoft's fault, check out the old DEC Rainbow. This was an MS-Dos machine that came out about the same time as the first PCs, but DEC set its demarcation point much higher – as did a couple of supposed IBM-compatibles (which weren't too compatible, as their owners discovered). Since IBM's PC hardware set the standard for all future MS-Dos computing, the decision has stuck if you want to use IBM compatible programs. The 640K point represents the boundary between free read-write RAM space (below), and system space (above).

Memory in a computer is best visualised as a single stack with addresses that extend from zero to the highest possible with the available processing chip – see Figure 1). And, the Intel 8086 line (which spawned the 8088 variation that IBM chose to use with the PC) has a 20-bit address bus, so the memory limit can be calculated as two raised to the power of 20, which is 1Mb (or 1,048,576 bytes, if you want to be pedantic).

Now, a computer can't make all of its addressable memory space available just for use by operating systems, applications and data. It also needs some space for mapping the video for the screen, and for other housekeeping functions usually stored in ROM. The video memory map obviously needs its own RAM because it holds the current image of the screen which changes constantly.

Memory addresses

THE POINT of all this is to clarify the fact that if you want to have 640K of usable RAM space in your PC or XT, then you need to have more RAM memory to support the video mapping. These chips may be on the motherboard, on an expansion card, or on a plug-in video card; it doesn't matter where they are physically, the computer will still address them as part of one single address map, from zero to the 1Mb limit.

So in the upper 384K of memory addresses in a PC you will find –

□ RAM chips reserved for the video mapping,

□ ROMs which control the hard disks and EGA displays,

□ A BIOS ROM,

□ Some free space, and

□ In the uppermost 64K (F segment), the Basic language in ROM.

When someone attempts to sell you a '640K PC', you need to ask whether the machine has the full 640K of 'conventional' read-write RAM space, or is that just a count of the number of RAM chips on board? Some retailers include the video memory RAM in with the count, so there's some variation in what you actually get with '640K compatibles'. Be warned!

According to IBM, the correct term for this bottom 640K of read-write user-RAM is 'conventional' memory, and it needs to be in a continuous run of addresses (contiguous). But despite this, it is not treated simply as a single chunk. To make it easier to work with memory, software programmers divide the total 1Mb of mapped space into sixteen segments, each of 64K. These are numbered from Seg 0 to Seg 9 for the first 10 segments (which makes up the total 640K of 'conventional' RAM space) then Seg A to Seg F, for the six segments used for video RAM and ROM above this to the 1Mb limit.

Just to emphasise one point: this 'system' space isn't totally occupied, there is usually at least one segment (64K) of free address space here, which can be used for expanded memory control.

Don't forget that the original PCs were released with only 64K of user memory, with four rows ('banks') of RAM, each row containing 16K with a parity bit. Later, 256K memory chips were used instead of the 16K ones, and this allowed the maximum memory on the motherboard to rise to 256K. Once the motherboard was fully populated, more RAM had to be added through the use or an expansion card in an I/O expansion slot.

There's another terminology trap here. 'Expansion' simply means anything (memory or any other type of card) which plugs into an 'expansion' slot – so with memory it can apply to normal expansion of the conventional memory (up to the 640K limit), or to expanded memory (bank-switched), or to extended memory (above the 1Mb range). Confusing, isn't it?

So, in the early days of PCs and XTs, expansion was still within the addressable limits of the processing chip (1Mb) – which meant user memory only up to 640K. At the time the PC was being designed, 640K sounded like such a lot of memory, but now it is a severe limitation. Fortunately, some fairly clever things have been done to get around the problem.

Expanded memory

THE CONVENTIONAL memory must house the operating system, device drivers and memory-resident TSRs (Terminate and Stay Resident programs) in the 640K space, along with the applications and the data. However, you can fool the processor into using more than this through some clever 'expanded memory' software techniques, as long as you have the extra memory chips on a special add-in board. So correctly, this should be termed an 'expansion card with expanded memory'.

The expanded memory technique used is straight bank-switching – a technique copied from the Apple II. The idea here is to exchange memory modules in 16K 'banks' (a quarter of a segment) between the conventional memory addresses and those non-addressed chips on the add-in expanded memory card. You can then use a software switch to drop these banks in and out of the normal memory map – substituting them temporarily for quarter-segments of conventional memory space.

Each bank on the add-in board has a register that specifies its required address, and the operation is controlled by a MMU (Memory Management Unit) on the card. When the program needs to use this expanded memory, it clears the contents of its current address register (saving it on a stack for later recall) and replaces it with those of the required bank(s). As a result of this 'duplicity', the CPU can be fooled into treating two (or more) different sets of memory banks (one on the motherboard and the others on the add-in card) as if they were only one, however, the ad-

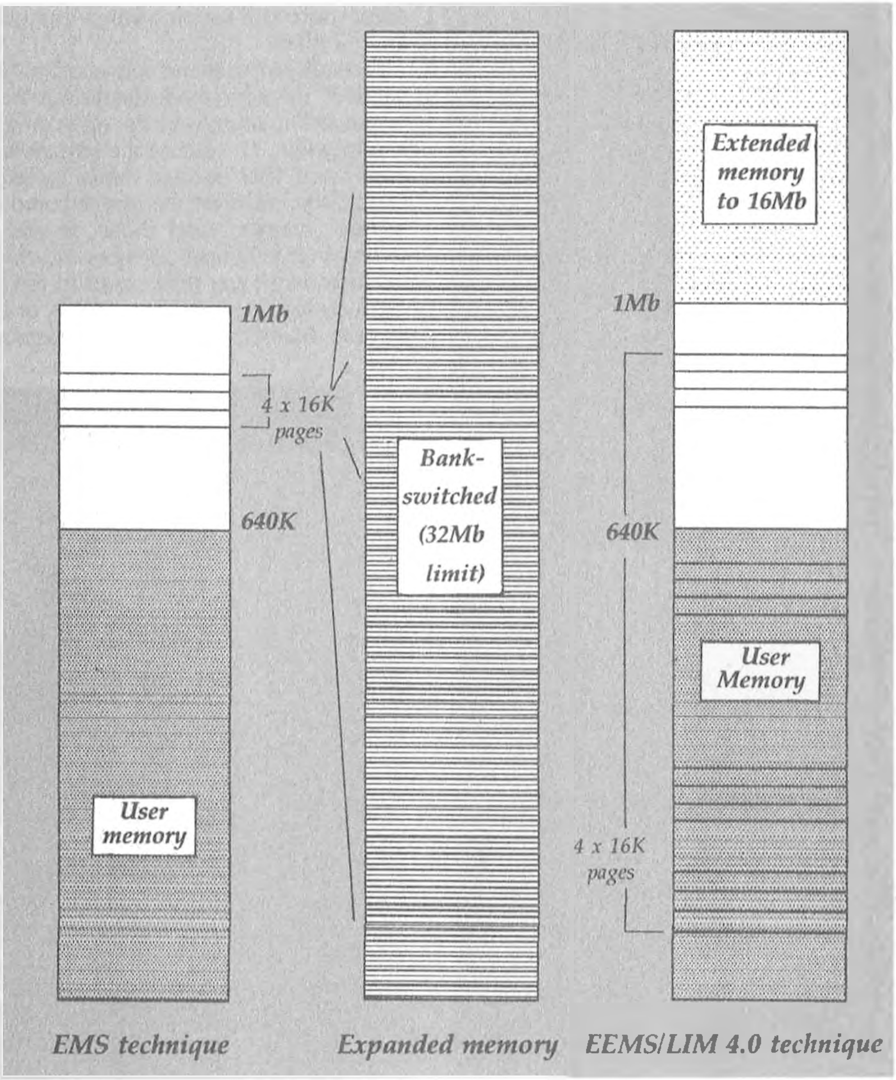


Figure 1. EMS systems swap 'pages' of memory in and out of four 16K pages of system memory (above the 640K limit) – EMS 3.2 supported 8Mb of expanded memory, while version 4.0 supports 32Mb. The term 'expanded memory' is now applied almost exclusively to EMS. EEMS is AST's super-set of EMS; it allows 64 pages of bank-switching.

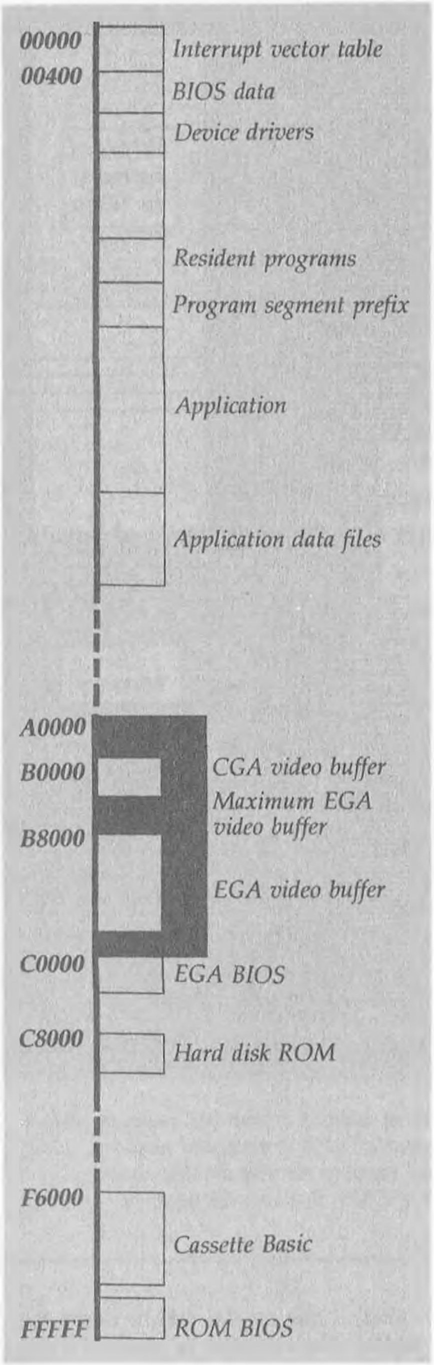


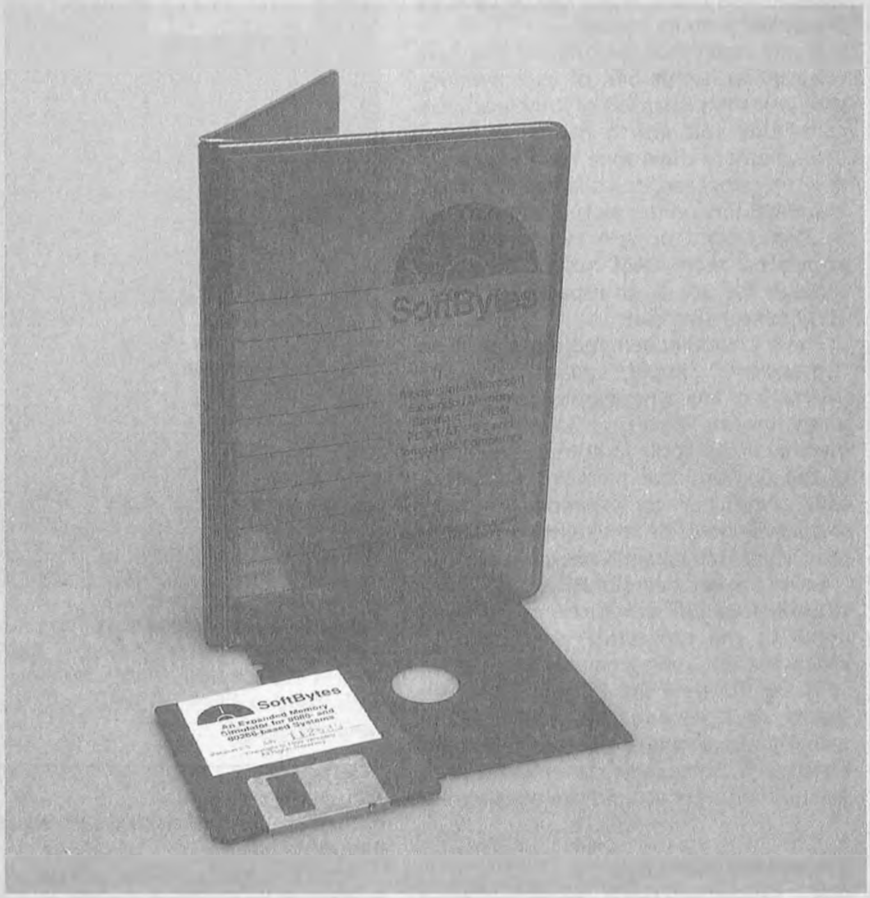
Figure 2. *How the 1Mb of address space is used in Intel's 8088 chip. Note that Dos doesn't have a 640K limit to the memory size – when designing the XT, IBM wanted to have a fixed location for their video RAM and placed it in the address space starting at 640K (A0000 hex in the diagram). The gaps in addresses between the blocks below A0000 are used in bank-switching. The Program segment prefix is a 256-byte header that Dos places before all .EXE and .COM files.*

dress space still remains within its logical limit of 640K.

So bank-switched memory is called 'expanded' memory to distinguish it from 'extended' memory – which we'll come to in a moment. To confuse the terminology even more, IBM becloud things by referring to the chips on the motherboard as 'planar' memory, and those on add-in cards as 'I/O channel' memory. I/O channel memory (if you think about it) can be conventional memory (extensions), or expanded memory, or extended memory. They really go out of their way to make computers difficult to learn about.

To round out this historic picture of 'expanded' memory: note that software could only take advantage of bank-switching and expanded memory when it was written with this possibility in mind. And since there were many ways of implementing bank-switching, the software needed to be matched to the requirements of the add-in memory board. These boards were not cheap because they needed costly MMUs in addition to the memory chips.

An alternative . . .


SoftBytes
SoftBytes
An Expanded Memory Simulator for 8086- and 80286-based Systems

IF YOU HAVE an AT or PS/2 model 50 or 60, there is an alternative to spending a thousand dollars or more on a memory board – simulate the extra memory. SoftBytes does just that: it simulates the presence of one or more LIM version 3.2 boards. Instead of storing the EMS data in banks on an expanded memory board, SoftBytes uses hard, or even floppy disk space, or that usually wasted 384K above the 640K limit.

Expanded memory boards come with drivers – usually called 'EMS managers' – which provide the interface between the board and application programs designed to use EMS. The drivers take care of 'mapping' available banks of memory on the board into the EMS page-frame (a

Without a single standard for bank switching, few software publishers were willing to gamble on writing software with requirements larger than the 640K limit because they couldn't be sure which add-in memory boards could be used. As a consequence, the hardware manufacturers didn't have software that would use their product, so it was a confused form of egg-and-poultry stalemate.

Then, when the popularity of TSR programs reached the point where a 100K of RAM was being permanently occupied by these accessories, the problem reached a crisis point.

Lotus and Intel both decided that they had to do something about bank-switching standardisation. Lotus was having to write multiple versions of their software to feed the growing hordes of expanded memory cards, and Intel liked the idea of selling more chips – it was in the DRAM business in those days. The result of their collaboration was EMS (Expanded Memory Specification) which allows up to 8Mb of memory. (*For a discussion on the different sorts of RAM, see 'FRAM – a RAM that doesn't forget' in our April issue.*)

Microsoft was quick to realise the advantages of EMS, and from Dos 3.2 on, it ensured that the operating system was compatible. So with Lotus, Intel and Microsoft behind it, LIM (Lotus, Intel, Microsoft) EMS had enough clout to become established as the de facto industry standard – but it wasn't without its challengers: there was a later AST 'super-set' version from AST, Quadram and Ashton-Tate.

Half of them didn't have a clue about the reason behind the PC's original 640K limit, or how extended and expanded memory differed – and the other half thought they knew, but had it wrong.

The LIM EMS specification assumes that at least one segment (64K) of memory space is free in the memory map of a PC or XT between the normal 640K conventional memory limit, and the 1Mb upper processing limit. Not all this space is occupied by video RAM, and the ROM chips. This spare 'window' page-frame holds four 'pages' of memory, each of 16K. These pages can each hold switched pages from the expanded memory on the card. The EMS scheme allows the software to shunt data or applications in 16K page amounts into this addressable space from an add-in memory card under the control of a Memory Management Unit, and ultimately, device drivers in the software.

The card can hold (and the software can use) a couple of megabytes of this expanded memory, which are simply called in as required; the switch is virtually instantaneous. This memory area is outside the normal 640K, so Dos is unable to access it without special code being written; similarly TSR programs can't use this space. While in the addressable memory space, the pages are treated as extensions to the conventional 640K. So any processor, from the 8088 up, can access and use these pages only if the operating system and software allow access to this system space – the CONFIG.SYS file is used to set up this function.

I must also mention AST's super-set of the EMS standard, known officially as ... would you believe? ... Enhanced Expanded Memory Specification (EEMS). This was promoted to take the bank-switching idea to new heights – to 64 pages rather than the four available with EMS 3.2. Luckily, the two groups eventually got together and the result is LIM 4.0 which incorporated some of the best AST ideas into the one EMS standard.

The AST/LIM 4.0 enhancement did not limit the number of pages available for bank-switching in the addressable space to only those four 16K pages. It allowed a much greater number to be dynamically assigned by the software and to exist in any part of the addressable space. This

Choosing extra memory

AS A ROUGH guide to choosing extra memory, follow these rules:

□ For slow speed machines (below 8MHz) on the bottom end of the range (PCs, XTs and slow ATs), buy an EMS add-in board and use the expanded memory system whenever possible.

□ With faster ATs, always build memory into the motherboard before you consider adding EMS on hardware. It is always faster to use EMS emulation software and the motherboard extended memory for EMS, than add-in expanded memory.

□ With '386 computers you should always add memory to the motherboard first, then by using proprietary 32-bit expansion units, before considering off-the-shelf add-ins. Avoid using 16-bit expansion memory cards with these computers.

64K contiguous block of memory in the address space accessible to Dos – see Figure 2 in the main story).

SoftBytes stores and retrieves data from a disk or from extended memory and can simulate up to 8Mb of LIM/EMS – it does it easily – installation took only took several minutes. If you've got memory problems with the likes of 1-2-3, Framework, Quattro, Q&A or WordPerfect, at $149 this could be the cost effective solution. The program itself uses about 70K of conventional memory when activated and only 5K when deactivated.

However, like most simulations it's not perfect. For example, it doesn't allow page-aliasing – that's the mapping of the same logical page (bank of expanded memory) into more than one physical page. This means it can't be used with Javelin, Lotus HAL or other programs that need aliasing.

The page is slower than an expanded memory board – this may or may not be a problem, depending on how the program triggers the mapping and how it uses the expanded memory. The SoftBytes manual points out that some software will actually run faster without simulated EMS, than with it, because of the way it handles page mapping – Paradox was the example.

Ideally, you should take a copy of the application you want to try with SoftBytes' simulated memory with you to a dealer and try it before you buy – our copy came from Software Express: there are offices in Sydney, (02) 519 3249, and Melbourne, (03) 663 6580.

As a bonus with the simulated memory, a print spooler is supplied on the program disk. This can be a godsend if you need to print large files and would like to go on using the computer. For example, we 'spooled' a 166K file: printing it took over half an hour, but use of the computer was lost for less than a minute

Virtual memory

THE '286 AND '386 machines are capable of supporting gigabytes of 'virtual memory'. This is a software function made possible by the MMU working in conjunction with a modified operating system. Virtual memory allows the computer to fake enormous memory capacity, by allowing programs to ignore the distinction between RAM and disk storage. All the available RAM, and all the available disk space can be written to, and read from, at will.

The disk-based memory is divided into 'frames', that are swapped in and out of RAM as needed. Usually the frame will be the same size as a segment (64K on the '286), and the MMU will keep track of what frames are currently residing in RAM and where the others are on the disk.

The operating system must be capable of making decisions as to which are the least-needed frames in RAM, and instructing that these be copied back to disk to make room for new frames that are needed Often these decisions are made on the basis of 'clock algorithms' which drop a RAM frame if it hasn't been used for some time.

meant that these pages could be switched in both above and below the 640K barrier, thus making the system far more efficient and flexible, especially for multi-tasking and multi-user systems. Finally, the combined set of expanded and enhanced-expanded standards for memory are now known as E/EMS, just to add to the confusion.

So much for expanded and enhanced-expanded memory systems which are all based around the limitations of the old 8088 chip with its 20 address bus lines and its 1Mb peak address. The next subject in this cognitive web of semantic confusion, is the extended memory system, which came about because later Intel chips that evolved were able to address more than a single megabyte.

When the IBM AT made its appearance (powered by Intel's 80286 chip), the IBM PC family jumped from 20 to 24 address bus paths, and therefore, memory limits rose to a maximum of 16Mb (two raised to the power of 24). What is more, the AT's microprocessor was able to operate in two modes: the 'real' mode which imitated the standard PC and ran conventional MS-Dos and Dos-based applications, and the 'protected' mode. However, for a long time there were no applications written specifically for the '286's protected mode – except if you were interested in Xenix – so most ATs have always been employed as super-Dos PCs in the real mode.

Extended memory

IN THESE '286 machines, the memory space above the old 1Mb limit became known as 'extended memory' (which it was from the real mode's point of view – but not really from that of the protected mode). If you intend to use your machine to run OS/2 or programs such as Oracle (which have been specifically written for the protected mode, then this 'extended memory' is simply treated as a continuation of the first megabyte.

In this real mode/Dos-emulation environment, any read-write RAM above the normal 640K limit (but with addresses above the 1Mb range) could only be used as a print or disk spooler, or as a RAM disk. It was a cache area into which data was put when it wasn't actually needed by the processor.

According to IBM, the correct term for this bottom 640K of read-write user-RAM is 'conventional' memory.

There is one seeming exception to this: the XMS (eXtended Memory Specification) devised by AST, Intel, Microsoft and Lotus which allowed Dos to use 64K of 'high-memory' (above 1Mb) extended space as an extended read-write area, for a total of 704K. Windows/286 seems to be the only current program to recognise this extension, but it is available for others.

The XMS extended memory specification is a complex piece of memory map manipulation which allows addresses above the top end of the normal PC range to wrap around and appear at the bottom of the memory map, but 'only on machines having an A-20 address line', I am told. (I believe them!)

The difference between expanded and extended memory is supposed to be that the former does not have specific memory addresses. Expanded memory gets added to the system page by page, and assumes the address of the page-slot into which it is added. Extended memory has its own specific addresses above the 1Mb limit.

I don't understand XMS and it isn't important anyway, so I won't go any further. It appears to me to be a complex form of bank-switching, but Microsoft insist that it is extended memory.

Whatever! The software needs to be able to handle this rule-breaking if it is to take advantage of these rather minor extensions. You may wonder whether it was worth the bother, but it was at the time. Are you still with me? Unfortunately, it gets a trifle more complex yet!

The point is that '286 and '386 machines can have addressable memory above the 1Mb limit, but if they are running Dos and standard Dos applications in their 'real' modes (or in the 'virtual 8086' mode of the '386), the Dos can't see or use more than the 1Mb.

Extended memory with '286-based computers can go as high as 16Mb, and with the '386, as high as 4 gigabytes. And, both '286 and '386 computers can use both their extended memories and the old bank-switched expanded memories, if the application has been written to take account of this. This is what makes it all so confusing. An add-in expanded (bank-switched) memory card can also double as a normal extension of memory in these later computers (although you are wasting the MMU chip) but, with the exception of the disk and print spooling noted above (and XMS), you can't use this extended space in the 'real' mode. It is not for standard Dos applications.

Well, now! Suppose you have a '286 or '386 machine with a couple of megabytes, and you always use it for Dos applications. Since you've got all this extra memory unused on the motherboard in the 'real' mode, why should we need to add a special memory card to run the bank-switching expanded memory system? Why add new chips when we've got some sitting idle?

The answer is that you need a MMU (Memory Management Unit) to handle the bank-switching. The '286 doesn't have one that can handle this operation unless it is specifically provided on the add-in card – but fortunately, the '386 has such an MMU built-in. The '386 microprocessor can change the address of any bank of memory in the system. Therefore, with the

Memories are made of . . .

TO HELP sort out the confusion in terminology, here's a brief definition of the commonly used expressions –

EEMS: Enhanced Expanded Memory System – AST's super-set of EMS; it allows 64 pages of bank-switching, rather than the four available with EMS 3.2. The two were combined into AST/LIM 4.0 which allows pages to be switched in both above and below the 640K barrier (see Expanded memory).

E/EMS: Expanded/Enhanced Memory System – a general expression for all standardised forms of Dos memory management.

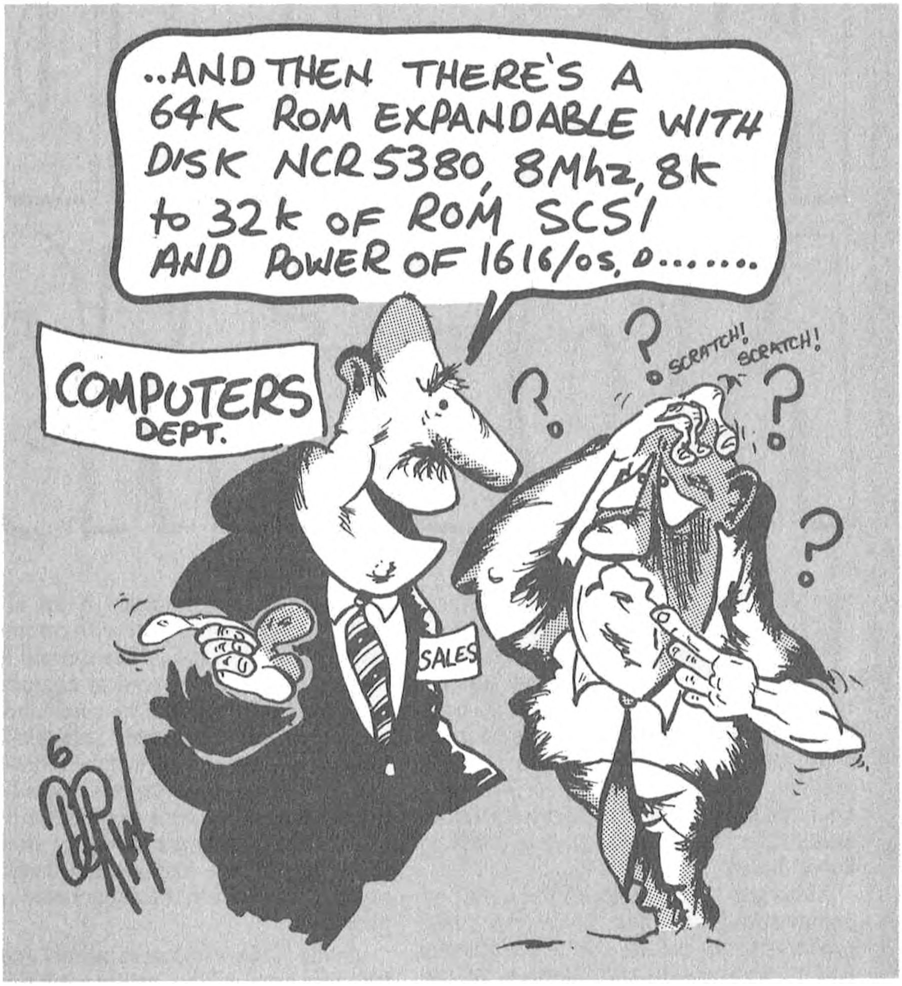
EMS: Expanded Memory Specification – the Lotus, Intel, Microsoft (LIM) collaboration which allows up to 8Mb of memory.

XMS: eXtended Memory System – devised by AST Research, Intel and Microsoft to allow Dos to use 64K of extended (not expanded) memory for a total of 704K. Windows/286 was the first program to recognise XMS.

Expanded memory: Originally a general term for a number of bank-switching techniques, but now it is applied almost exclusively to EMS. The LIM group devised this standard way of expanding the 640K limit. EMS systems swap 'pages' of memory in and out of four 16K pages of system memory (above the 640K limit). Not every program can use EMS because its sophisticated form of bank switching requires the hardware and software to work together. EMS 3.2 supported 8Mb of expanded memory, while version 4.0 supports 32Mb.

Expansion memory: A very general term which refers only to chips on an adapter card without defining whether these are being used in the expanded or extended state.

Extended memory: The extra memory above Dos' 1Mb limit; it is straightforward additional memory for 80286 (AT-class) and '386 machines. Only a few Dos programs can use it; the most common examples are utilities and disk caches. This memory space is primarily available for the likes of OS/2 and Unix, with the processor running in protected mode.


..AND THEN THERE'S A 64K ROM EXPANDABLE WITH DISK NCR5380, 8Mhz, 8K to 32K OF ROM SCSI AND POWER OF 1616/os, o.......
COMPUTERS DEPT.
SALES
SCRATCH! SCRATCH!
?
?
?
?

The result is LIM 4.0 which incorporated some of the best AST ideas into the one EMS standard.

right software drivers and Dos, the extended memory of a '386 sitting unused above the old 1Mb limit, can now be made available for bank-switched expanded memory. Now extended memory is also expanded memory!

The way it does this is to temporarily switch the processor from real mode to protected mode using routines contained in the AT's BIOS chip. The device drivers used in the E/EMS expanded memory system must also be modified to perform this function. You can get Above Disk and several other drivers for this purpose. Both Above Disk and a software developer's product called 386/VMM allow programs to use extended RAM even greater than the four gigabyte limit by swapping into a virtual memory mode when it senses the end of RAM addresses, and using the hard disk as a natural extension.

Got that?

Just a final point. With '286 and '386 machines, when we talk about memory, we recognise that it exists in three blocks. The 1Mb machine you buy from your local retailer will have the normal conventional memory in the address space extending upwards from zero to 640K. Then the next 384K of address space (to the 1Mb point) will be reserved for the standard video and ROM functions, and finally, there will be 384K (to make up the 1Mb of user RAM) of 'hi-mem' extended memory occupying addresses from 1Mb up to about 1.4Mb.

However, you've got to be careful here. Some of the AT-clones, and many '386 systems, reserve for themselves some system space within this top 384K (supposedly 'extended memory') so you won't get your full 1Mb of usable RAM.

Now, you are allowed to go away and quietly beat your head against a wall! □

PCS AND MODEMS

I'VE JUST finished writing a user's guide for an Australian modem manufacturer and, quite frankly, although I use a modem pretty regularly, this has been a subject that is both confusing and difficult to come to grips with. Mostly, I suspect, this is because modem technical information has only been dealt out by manufacturers reluctantly (like ASIO and MI-5) on a 'need to know' basis.

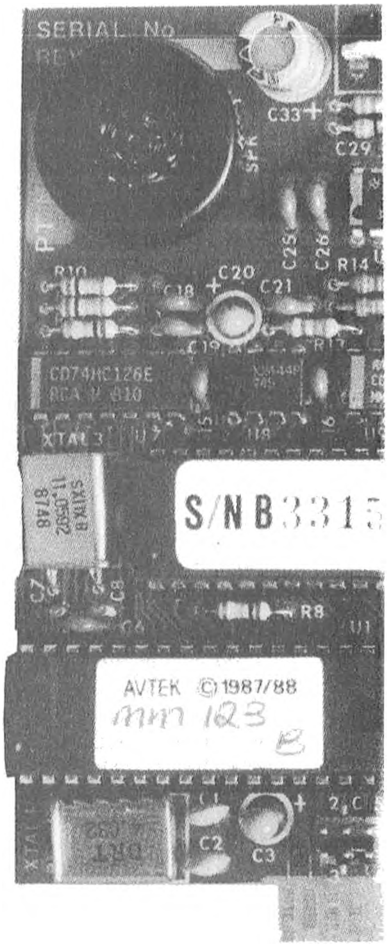
Although I've written quite a lot on communications and modems and modem-theory before, I've never actually had to look inside the workings of the command-set and worked out what is going on with intelligent modems; how they interact with their software – and why they sometimes don't. Now that I've had to, I'm going to bore you with the details. Anyone interested in communications will understand the frustrations of modems. How is it that everything can work perfectly one week, and not the next? Why can I chat over the keyboard with a friend, but we can't transfer files? Why is it that PC-to-PC, or PC-to-mainframe, or PC-to-packet-network can be so simple when everything goes right, and so excruciatingly frustrating when it doesn't?

So this is a two-part attempt to unscramble the inner workings of today's intelligent modems and understand how they are designed to operate with today's intelligent software. We won't be dealing with any particular brand of modem or software, but with the concepts that underlie the Hayes AT-command standards. We will also look at how various manufacturers have designed their equipment to use these AT-commands, and overcome the inherent inadequacies. A lot of the problems many of us have with communications comes about because we have over-emphasised the need to conform to the Hayes commands. This command-set became the de-facto world standard in a vacuum left by the failure of the rest of the world to agree. The Hayes standard has now grown as modems and software became more intelligent, but they haven't fixed some of the fundamental flaws because of the need to retain backward compatibility.

Ideally, if the world were perfect, manufacturers were willing, and modem buyers were logical, we would scrap the Hayes system and start again. But true 'standards' being what they are in this industry (a rare commodity, to be prized) any manufacturer that took this radical approach would probably be facing economic suicide – at the lower end of the modem market anyway. In America, Hayes have the modem market by the throat, so you'd never get world agreement on anything new.

Transmission Standards

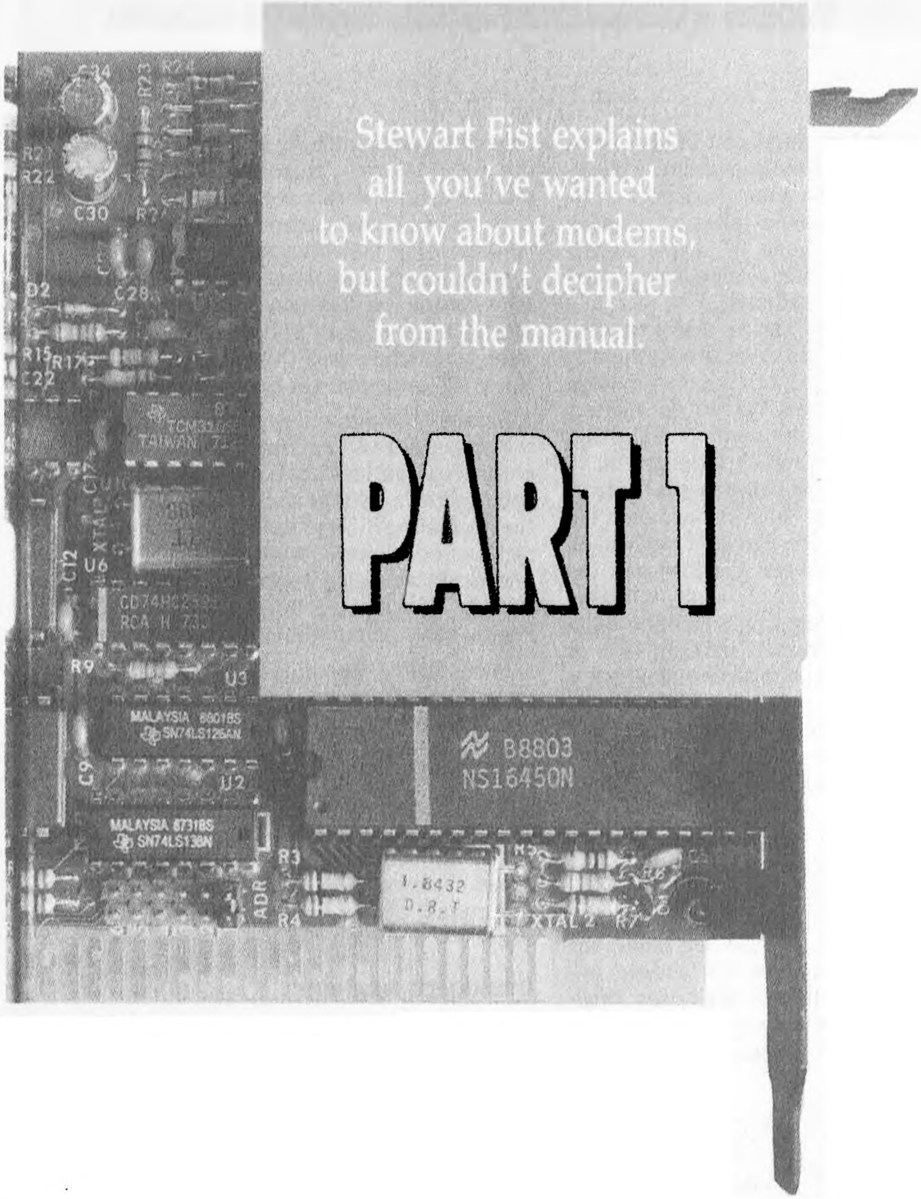
I SHOULD stress here that I am going to be talking about two-wire asynchronous modems of the type suitable for the normal switched telephone network. There are also all sorts of synchronous and four-wire modems for special purposes.



The significance of 'asynchronous' here is that the two parties don't first need to establish 'clock' timing to send the data from one to the other in a perfectly regular (synchronous) controlled manner. When you can do this, communications is often faster, which is why the synchronous approach is ideal for mainframe-to-mainframe links.

But if you are sending from a keyboard or transmitting short files with fairly primitive error-checking systems, the traffic will be 'bursty' and erratic – so asynchronous ('without synchronisation') is the obvious way to go. And you will not be alone; ninety-nine percent of modems would be primarily used for asynchronous communications.

Australia conforms to the transmission standards set by the CCITT (*Comite Consulatif Internationale de Telegraphiques et Telephonique*) used by most of the world except for America and a very few of its colonial cohorts. The Americans have a couple of modem transmission standards of which Bell is by far the most widely used. Fortunately, the Americans have recently decided to come into line with the rest of



Stewart Fist explains all you've wanted to know about modems, but couldn't decipher from the manual.

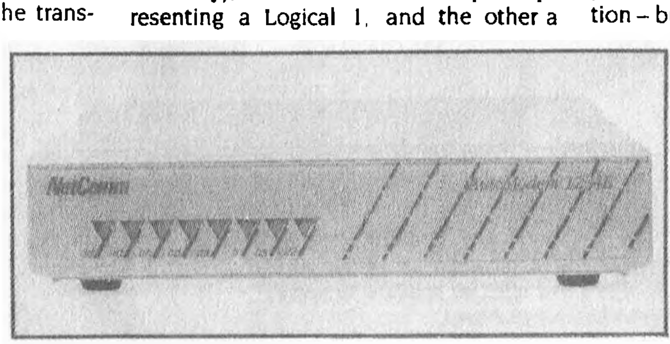
PART 1

the world. At the newer higher data-rates, that country now conforms to CCITT standards, but there are some oddities, like those that conform to CCITT at one data-rate, then fall back to Bell at another.

This article isn't about modem transmission technologies, rather it is about the control systems between your computer and its local modem. But we need to refresh your mind about what the transmission standards entail, so let's just have a quick look at the various CCITT and Bell data-rates.

Don't forget that with full-duplex (two-way) transmission there are audio frequency tones used by the Originator (usually the person who makes the call) while different tones are used by the Answerer. It doesn't matter who uses what tones, as long as they both don't both use the same.

300bps full-duplex. This CCITT standard is called V.21. Originally it was for only 200bps, but they later stretched it to 300bps. V.21 uses a simple type of Frequency Shift Keying (FSK) with four audio frequency carriers – two pairs (one pair each way); with one tone of each pair representing a Logical 1, and the other a Logical 0. The V.21 Originate frequencies are (0) 1650 Hz and (1) 1850 Hz and the Answer frequencies are (0) 980 Hz and (1) 1180 Hz. This swapping of tones gives V.21 its characteristic 'warbling' sound.



The American standard here is Bell 103 which is essentially the same as V.21, except that it uses different audio frequencies. Originate uses (0) 1070 Hz and (1) 1270 Hz, and Answer uses (0) 2025 Hz and (1) 2225 Hz.

1200bps full-duplex. The CCITT standard is called V.22. It uses Phase Shift Keying (PSK) – where an audio frequency is taken 'out-of-step' by a set amount. The V.22 standard uses four possible 'phase changes' (by 90 degrees each) and so each change can carry two bits of information (dibit) which is where the relationship of 'baud' (a change) to 'bits-per-second' fails. Each change is one 'baud' but it provides two 'bits' of information.

Bell 212A is the American equivalent which uses different originate and answer frequencies.

1200bps half-duplex and 1200/75 videotex. What makes this standard so confusing is that it began as a 600bps half-duplex standard, migrated to 1200bps half-duplex, then took on a 75bps 'back-channel' which made it a sort of hybrid – halfway between half-duplex and full-duplex. The data-rate limitations come about because it still uses the old FSK transmission techniques, and the bandwidth of the telephone channels don't allow full-duplex 1200bps transmission using FSK.

When a single pair of wires is being used, the telephone channel bandwidth cuts off frequencies above about 3.5 kHz. In half-duplex FSK, only one frequency ever occupies the line at any one time – but with full-duplex both may be transmitting, and the peak bandwidth requirement must be calculated by summing the highest frequencies.

So, if we need the highest audio tone of a pair to be about 2100Hz (about twice the bit-rate of the 1200bps signal) the telephone line can handle that in one direction – but it can't handle two-way simultaneously. This would mean a guaranteed delivery of 4.2kHz, and most of our telephone lines only guarantee about 3.2kHz.

They compromised with this 'hybrid full-duplex' standard ('split-baud') for videotex, by lowering the 'back-channel' data-rate to 75bps (which could be carried on a 450Hz signal). The sum of the two frequencies is now only 2550Hz, which is well within the 3.2kHz limit. The compromise works well; the slow 75bps rate is ideal for sending keyboard commands while the 1200bps return rate is ideal for videotex screens and file downloading.

The CCITT standard here is called V.23 and it uses FSK transmissions with audio frequencies of Originate (0) 390Hz and (1) 450Hz, and Answer (0) 1300Hz and (1) 2100Hz.

These modems were originally 600bps, and you'll still find some advertised as 'V.23 modems' which will only provide 600bps half-duplex – so be warned! However most of the newer models can now be used as either a videotex 1200/75 or a half-duplex 1200bps modem: the CCITT now differentiates the Mode 1 (600 half) and the Mode 2 (1200 half, or 1200/75) operations.

There is no Bell equivalent to the videotex 1200/75 standard, but Bell 202 is a half-duplex 1200bps standard (from 600 up to 1800bps) with a 5bps back-channel.

2400bps full-duplex. In the CCITT camp this is called V.22bis (the 'bis' being French for 'twice'). It uses a combined PSK and Amplitude Modulation (AM) technique called Quadrature Amplitude Modulation (QAM) which, naturally, is made up of 4 phases and 2 amplitudes.

The American equivalent is Bell 2400 which is the same as V.22, but it is usually implemented with a fall-back (when the lines are bad) to the Bell 212A standard.

The CCITT (and I assume the Bell) standards also define an 'echo-disable tone' which can be used to switch off the echo-suppressors at the ends of trunk lines. These suppressors are used by telephone companies on long-distance circuits wherever four-wire networks are connected to two-wire systems. Suppressors often cause modem problems, but don't confuse this form of undesirable echo with the full-duplex/Echoplex technique.

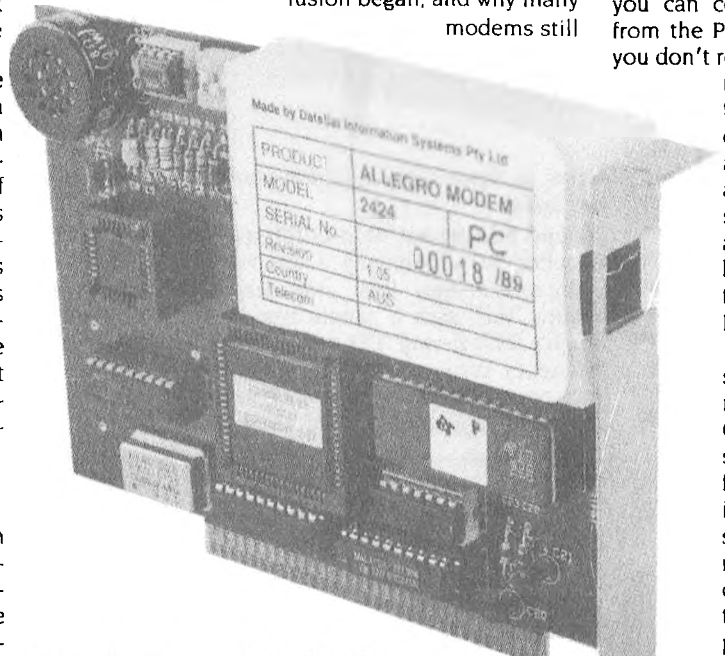
Signals

WHEN THESE transmission standards were being formulated in the early days of computer communications, the modems were entirely hardware devices, so there weren't any software commands. They took a cable feed of DC digital signals from the serial port of your computer or dumb terminal, and simply piggy-backed them onto audio frequency carriers so that they could travel for distance down the phone lines. DC digital signals will only travel along a cable uncorrupted for fifty to a hundred feet.

Everyone keeps saying that a modem 'takes digital signals and converts them to analog', but that's not correct. A modem takes digital signals carried on a direct current, and converts them to digital signals carried on an audio frequency. You've got to distinguish between the 'signal' which carries the information, and the 'carrier' which is the transport mechanism – distinguish between the load and the cart.

The early modems only had one data-rate and one 'national' standard (Bell or CCITT). So the box had an On/Off switch, a Talk/Data switch (to let you dial and perhaps talk to the other party) and an Originate/Answer switch that defined which of the frequency pairs you would use.

Not all of the early modems had this Originate/Answer switch, some assumed they would always 'originate' the data (mostly those attached to mainframe databases) so you could call them, but you always had to use the 'answer' frequencies (normally the caller uses 'originate'). This is where some of the later confusion began, and why many modems still have a manual switch for this decision; if the caller always 'originates' then the modem can make this decision for you.



Internal modems – whether for PCs or portables offer a cost effective solution to many communications problems.

When modems began to get basic 'smarts' – like the ability to answer an incoming call, or to dial out for you (rather than relying on a separate handset for dialing) – they began to need some software commands, and this is where the Hayes AT-command set began.

Simple answering involves detecting the ring-tones (AC currents) on the line, picking up the line (taking it 'off-hook'), and sending a burst of carrier tone so that the originating modem will know that contact has been made. When there is only one data-rate being supported, there is nothing complicated here, but it does need a basic amount of intelligence.

Pulse dialing (as we use with the older rotary phones) simply involves taking the line 'off-hook', waiting a few seconds for dial tone, then shorting across the DC voltage on the lines, a set number of pulses (for each number), with the correct amount of pause time between each number in the sequence. The standard is 10 pulses a second (50 millisecond, breaks and makes) and an 'Interdigit Interval' of about 700 milliseconds.

Intelligent modems

AT THIS LEVEL of modem automation you can control most of the functions from the PC's communications software; you don't really need much intelligence or memory in the modem. But as things become more sophisticated, with multiple data-rates and chips that can recognise and generate both the Bell standards and the CCITT standards, it becomes imperative to build a small computer into the modem with its own RAM, ROM and often an EEPROM.

So the next command-design step was to provide the modem with two states – the Command-state, where you send it commands and it performs the dialing and answering functions, and the Data-state, which is the on-line mode where it transmits any data sent to it on through the telephone lines – and where it passes anything received, back to the PC.

If you are going to have two states, then you've got to have

Dialing commands

NOT ALL modems have all of these dialing commands, but most will have those listed as far as the 'R' command. Generally dialing commands are sent to the modem as one string, with the 'ATD' being the command sequence at the head of the string, and the remainder being parameters which are temporarily stored

ATD – 'Attention; dial' (this is the most basic command)

P – use pulse dialing (the default; it must precede the number to be pulse-dialed).

T – use tone dialing (this can also be used in a string to change from pulse to tone mid-stream).

, – the comma causes a pause in the dialing sequence, usually of two seconds.

; – the semi-colon causes the modem to return immediately to the Command-state after dialing (it must be at the end of the string).

R – use the 'Answer' frequencies rather than the 'Originate' ones (it must be at the end of the string)

W – wait for up to 30 seconds for the dial-tone before dialing (most modems can't detect dial-tone so a simple pause is used; it must precede the number sequence)

@ listen usually for 5 seconds of silence, then dial (if it detects a busy signal, it disconnects)

S=n – dial a stored number when it follows a standard 'ATDT' dial command (the number has previously been stored in location 'n' by the 'AT&Zn' command)

L – dials the last dial string again. (This command is also duplicated by 'A/' which is used without a preceding 'AT' or followed by a return; it causes the last dallying sequence to be repeated).

some way of switching between them – and since hardware switching is not conducive to computer automation procedures, you need identifiable software commands. At this point the AT-command set begins to be needed.

You won't often see these commands in action nowadays because the intelligent software takes over the job of sending the correct instructions to the modem – you simply make a menu selection. But behind the scenes (or if you are in Terminal Mode in your software) this is what happens: The Command-state to Data-state switch is easy; Hayes decided to make it 'ATO' (capital O for On-line); the 'AT' simply stands for 'Attention', and it is their way of telling the microprocessor to perform some function – in this case to store a value of zero, in the variable named 'O' in RAM.

Now this isn't readily apparent at first, because the zero has been assumed, but basically the command 'ATO' is the direct equivalent of the Basic statement *LET O = 0*, or to put it in English *Let the variable named O, now store the parameter/value 0*. You can use caps or lower-case in most modems, but some require caps only.

Whenever the modem's microprocessor now looks at the variable O, it will read the parameter 0 (zero) and know to switch (or stay switched) to the on-line Data-state. Reversing this process isn't so easy. It would be nice if we could send the command 'ATO1' or 'ATO=1' to tell our now-slightly-smart modem to switch back from Data-state to Command-state. The problem is that in the on-line state, the modem would simply transmit the command on to the remote modem as data. How can it decide that the occasional 'AT' is not the word 'at', but actually a command?

So the command to change back (called the Escape Sequence), involves both a pause-time and a unique set of characters. You've got to pause for about one second, then type '+++' (three pluses in sequence), then pause for another second. It doesn't need a return, and it is one of only two commands in the set that doesn't need to be preceded by the 'AT'.

It is assumed that these Escape sequence conditions are so unusual in the data stream that they can only mean 'Swap back to Command-state'. Actually, in most modems, you can change the 'guard-time' (before and after pause) and the characters you use (default is the +++) for this Escape sequence, but I can't imagine that you would ever need to.

Since we now have the modem recognising 'AT' as a command meaning 'Store' or 'Act upon', we can now construct a dialing sequence which will be held within the modem, and run under the control of the modem's microprocessor – rather than needing to send it numbers, one at a time, from the PC.

This is the 'D' (for Dial) command. So if you send 'ATD' followed by six or seven numbers, the modem will pick up the phone and dial (using pulses) the number sent to it and stored in RAM. Again, there is a variable implied here – the use of Pulse dialing has been assumed as the default. The correct sequence would be 'ATDP*xxxxxx*'.

Tone dialing

TONE DIALING came a bit later. This required the modem chip to be able to generate the 16 dual-tones (DTMF) which are fortunately world standards for telephone systems. The Hayes command to dial using tone is simply T instead of P: so ATDT*xxxxxx* will work the same as the above, except for tone dialing.

In most modems you can set the (temporary) default to Tone simply by sending the command 'ATT', and then reset it to Pulse with the 'ATP' command.

Many business travellers have found fax/modem combinations the most flexible means of staying in touch with the office.

The point to note here, is that the number to dial and the 'P' or 'T' variable parameters are all being temporarily stored in RAM, because this is the way that very simple microprocessors operate. They can only work on certain active areas of memory (RAM), and they only function by checking the parameters (values) stored in certain labeled memory areas (called variables).

I'm not going to go into all the commands in the Hayes AT-set here; you'll find them in your manual. The point of this article is to get some concept of what is happening inside the modem, and why it operates as it does. The modem will have a large number of variables held in

RAM, and whenever it needs to check out what to do, it will go to one of these variables, read the parameter stored there, and make a decision accordingly.

How did all these variables and parameters get into RAM in the first place? We certainly don't load them in by keyboard every time.

Intelligent modems will all have a range of variable-parameter pairs stored in ROM (the factory defaults). There might be about thirty of these, all labeled by a single-alpha variable name like 'M' for control of the loudspeaker 'monitor', or 'V' which controls the response and error commands on your screen. Each of these will hold a parameter (a value) which defines how the modem acts and which is changeable.

There are also 'registers' which are simply arrays. These are just variables with names like 'S3' or 'S9'. These hold parameters which perform simple housekeeping chores like setting the time needed to pause before and after the "+++" sequence, and how many rings before the modem should pick up the phone when answering.

When you first boot up your modem, it will transfer all these default variable-parameter pairs from ROM into the appropriate area of RAM (the active part of memory). So you can later over-write these variables to make temporary (or semi-permanent changes) if you want your modem to vary from the way it was set-up in the factory.

A good 'intelligent' modem will also have some 'non-volatile RAM' (battery backed RAM) or an EEPROM (Electrically Erasable Programmable ROM) which can hold these variable-parameter pairs when the power to the modem is switched off. So you can make changes from your keyboard into RAM, then instruct these to be stored in semi-permanent memory in the EEPROM. These will then be restored (loaded into RAM) automatically the next time you boot up.

Conceptually, the boot up process will then involve three steps.

1) Load all factory-default variables from ROM into the active area of RAM;

2) Overwrite some of these variables in RAM with any stored in the EEPROM (your semi-permanent modifications)·

3) Overwrite any of these with specific commands that now come down to the modem from the keyboard or from the intelligent software in the Command-state (your temporary modifications). It's a good system, but it doesn't take care of all the variables, because Hayes has left a lot of gaps in their command set standard.

For instance, you'd expect there to be a variable which tells the modem to use a certain data-rate. Common-sense says that there should be a command that sends the modem a parameter of 1 if you

Basic concepts

THE BASIC concepts and terminology behind data communications are straightforward, it's just that the jargon can be overwhelming to new users.

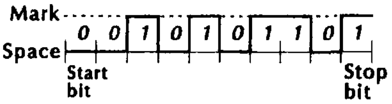
Signals and data passing between a sender and receiver follow a path through any number of physical components (the PC, modem, telephone lines, the receiving modem and so on), so the term *channel* is used to describe the logical path. If data is only ever sent in one direction (user A to B), this is a *simplex* channel. When communication can be in both directions, but not simultaneously (A to B, then B to A) the same channel can be used; this is a *half-duplex* channel. If signals or data are to be sent simultaneously, then two parallel simplex channels are needed; this is a *full-duplex* channel – these do not always need separate physical paths (see below).

Data communications is concerned with the transmission of binary digits, *bits*, that comprise the information being sent. These bits can be sent one after another, *serially*, or, using a cable with a number of wires, that number bits can be sent in *parallel*. Because of the cost of the cabling, parallel transmissions are only used for short distances (from a PC to a printer is the most common example); another reason that serial transmissions are the most common is that the phone lines don't have enough wires to support parallel transmissions (this is again, because of the cost).

Serial transmissions

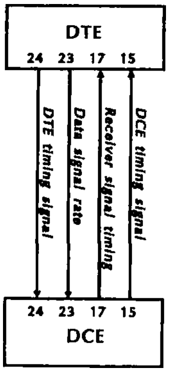
SERIAL transmissions can be performed in a variety of ways depending on the type of devices that are communicating (modems, terminals, whatever), but in all cases a *mark*, usually a negative voltage, representing the binary '1', and a *space*, usually a positive voltage, representing '0' are transmitted.

Asynchronous transmissions: Many devices are character (byte) oriented; since it is not possible to predict the number of bytes that are to be transmitted or received at any time, each one is dealt with individually without regard to time, that is, asynchronously. When no byte is being sent, the channel is kept in the mark condition; this is known as the *start bit*. After the data is sent, one or two extra bits are added; these are the *stop bits* that tell the channel to return to the mark condition so the next start bit can be recognized. (Mechanical devices usually require two stop bits, and electronic ones, one.) This is how the asynchronous transmission of '01101010' (the letter 'j') looks –



(Note that it is sent backwards, that is, the least significant bit first. I'm not sure why this is so, but it doesn't matter if both the sending and receiving devices are aware of it.)

Synchronous transmissions: If a lot of data is to be sent, the asynchronous method is inefficient because the start and stop bits carry no data, but still need to be sent. Synchronous transmissions overcome this by sending a timing signal that is used to synchronise the sending and receiving devices. The clock providing the timing signal can be either in the DCE – 'Data Circuit terminating Equipment' such as modems – or in the DTE – 'Data Terminal Equipment', such as computers, although it is usually in the DCE. (As you might guess, the nomenclature dates from the 1930s.) Some DCEs have a two-speed clock, so a data signal rate selector is set for the higher speed and un-set for the lower. The asynchronous connection looks like this –



The numbers in the diagram refer to pin numbers in the standard 25-pin RS232 serial port connector. Note that, while there are four lines in the connection, only three are used; which three depends on where the clock signal originates.

– Jake Kennedy

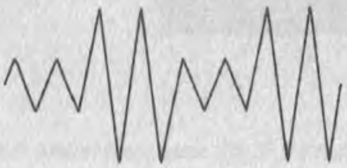
Mo-Dem

THE SIMPLEST form of transmission is to just send the digital signals down the wire. This is what happens with the simplest connections, such as a null modem, where two devices are connected directly – this is referred to as *baseband* transmission since no carrier signal is used. Since such a signal will drop to almost nothing over any useful distance, the only real viable alternative is to use the telephone network with its builtin boosters.

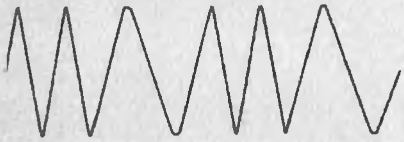
However, telephone equipment is designed for voice transmissions, not digital. The solution is to overlay the digital signal with a continuously varying AC signal – the technique is referred to as *modulation*. (Stick with us, the following looks more complex than it is.) A sine wave can be expressed with the general formula –

$h(t) = a \sin(ft + p)$

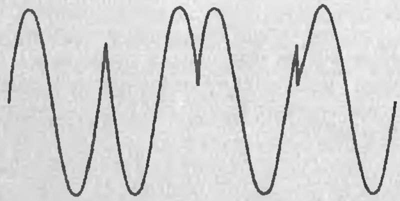
Here, **a** is the amplitude of the wave, **f** is the frequency and **p** is the phase. Any or all of these three can be varied (modulated) while time (**t**) remains constant. This variation can be detected by the receiving device and *demodulated*. Hence we have the name 'modem' – MOdulator-DEModulator. Amplitude modulation cannot usually be used for data transmissions because the signal becomes garbled over a phone line, but this is how the wave would look –



In frequency modulation (FM – and this is the same FM of radio transmissions, but it is more often referred to as 'frequency shift keying', or FSK, in data communications) two or more tones are used to represent digital values. This method is used for 300 and 1200bps modems. The wave looks like this –



In phase modulation, the phase is shifted $\pi/4$, $3\pi/4$, $5\pi pi/4$ or $7\pi/4$ giving four possible data values –



Phase modulation is used in 2400bps modems.

In order to transmit data using using either of these methods, a range of frequencies around that one being modulated needs to be available – this range is the *bandwidth* of the channel; in general, the higher the bandwidth, the higher the possible rate of transmission. Now, when the telephone line – or other cabling – is 'active', it has its own bandwidth, which is much broader than the channel bandwidth. This means that more than one channel can exist within a single telephone line. And that's the secret to full-duplex transmissions, which use two channels simultaneously, and many synchronous transmissions, which require an extra channel for the timing information.

– Jake Kennedy

want 300bps, 2 if you want 1200/75, 3 if you want 1200bps and so on.

This would be the intelligent approach, but that variable and command doesn't exist – because the early Hayes modems all had manual switches!

You'd also expect there to be a variable that tells the modem to use the Bell standards, rather than the CCITT. But there isn't, because all the early machines were either one or the other!

You can easily see that the command-set standard 'growed' by itself; it was never planned.

Another area that creates problems for occasional (or first time) modem users, is the difficulty in differentiating between commands that need to be sent to the modem in a communications session, and those that are intended for use by the computer itself.

The selection of terminal-emulation type (normally TTY – which stands for teletypewriter, so you can see how far back this goes) is obviously a computer problem, but some others aren't so obvious. Is it the computer or the modem which makes decisions about the use of 7 or 8 bits, or the number of stop-bits, or whether to use Odd, Even or No Parity?

It turns out that these are all computer functions. In fact they are performed in the UART chip which controls the serial port at the back-end of the PC. The modem only takes a flow of bits given to it in the on-line Data-state and passes them on: it doesn't care how many bits there are, what groupings they are in, and it hasn't got a clue about parity. (This is not true of the latest batch of modems with data compression and MNP error-checking)

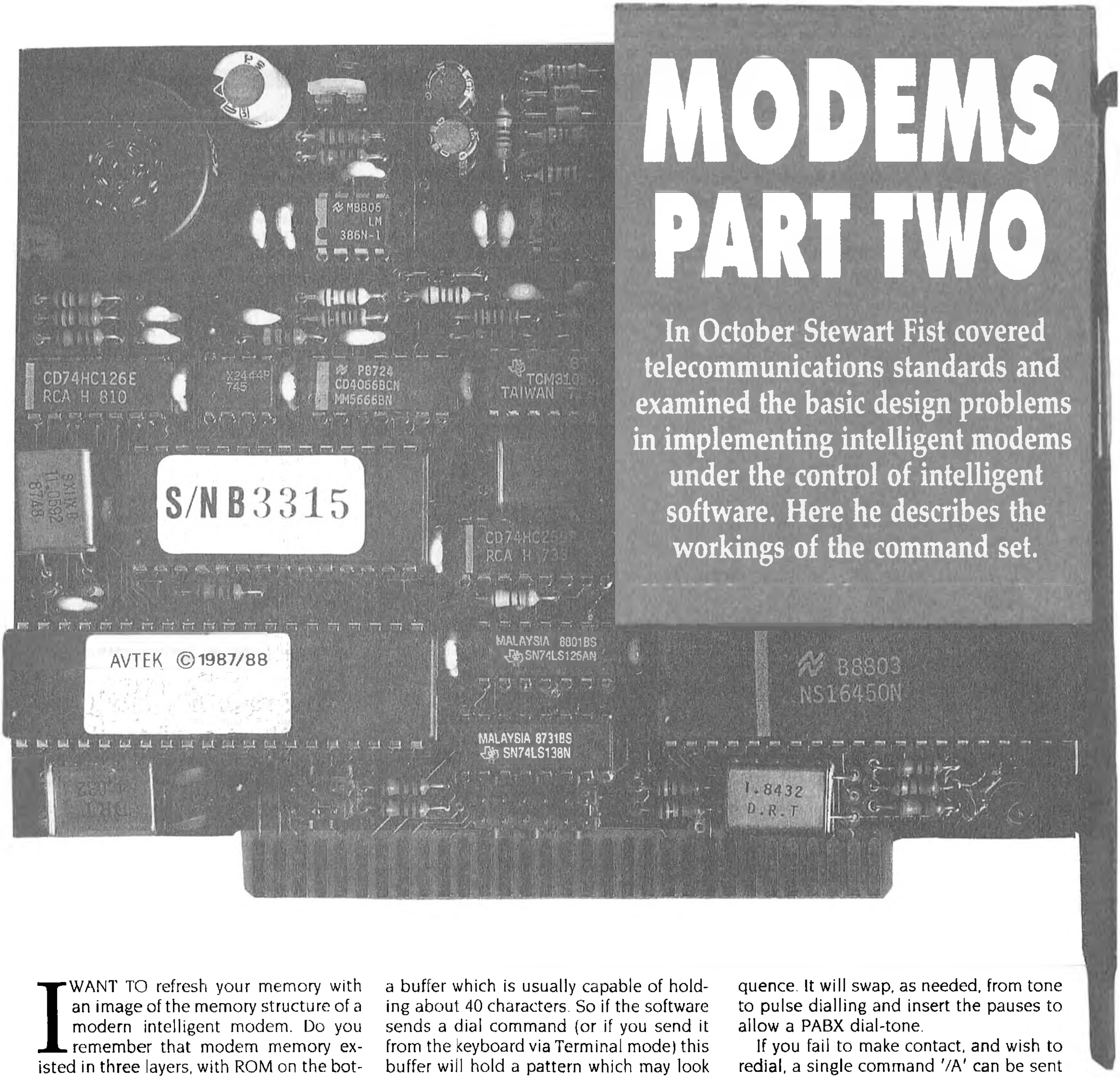
There is also a common misconception about 'buffers'. Most modems don't have buffers, so what comes in at one side goes out at the other instantly. The XON-XOFF type of flow-control is a function of the computers at each end, not the modems. So you won't find any 'AT' modem commands for this kind of flow control.

Similarly (except for the new MNP protocols) error-checking systems like XModem, YModem and Kermit are end-to-end functions that are performed by the computers, and have nothing to do with the modems.

Next month we will look at the way data-rates, national standards, and set-up procedures are communicated to the modem, and how it stores and acts on a range of parameters.

This complexity is one of the reasons why people find it so hard to decide what to do with communications software when things go wrong, or when they step out of the ordinary. There are so many choices – and since the modem manufacturers don't think that most of these are any of their business (they are software problems), they usually don't bother to mention them in the manuals.

Next month we will look at the way data-rates, national standards, and set-up procedures are communicated to the modem, and how it stores and acts on a range of parameters. Hopefully this will give you a conceptual grasp of the process, and some insights into the reasons why you have those erratic and 'unexplained' problems. □



MODEMS PART TWO

In October Stewart Fist covered telecommunications standards and examined the basic design problems in implementing intelligent modems under the control of intelligent software. Here he describes the workings of the command set.

I WANT TO refresh your memory with an image of the memory structure of a modern intelligent modem. Do you remember that modem memory existed in three layers, with ROM on the bottom, EEPROM above, and RAM at the top?

When you first boot up your modem, it will transfer all the factory defaults from ROM into the appropriate area of RAM (the active part of memory). Intelligent modems will also have some 'non-volatile RAM' (battery backed RAM) or an EEPROM (Electrically Erasable Programmable ROM) which can hold changes to these factory defaults (semi-permanent defaults) when the power to the modem is switched off. You create these changes by instructing current values in the modem buffer to be stored in the EEPROM with an '&W' (Write to EEPROM) command.

Figure 1 shows the process involved on boot up. Note that Phase 3 takes place via a buffer which is usually capable of holding about 40 characters. So if the software sends a dial command (or if you send it from the keyboard via Terminal mode) this buffer will hold a pattern which may look like this: 'ATDT1234567' ('Attention: dial using tone the number 123 4567').

It could be a much more complex series of commands that include parameter changes also, such as a sequence like this: 'ATB2M0DT0,,P12345676' ('Attention: change temporarily to the B2-rank, switch the monitor/loudspeaker off, dial using tone the number zero, wait four seconds, then dial using pulse the number 123 4567').

When the modem's microcomputer looks at the buffer, it sees the 'AT' command and knows that it must act by making these parameter changes in RAM (changing the B-variable to 2, the M-variable to 0) and then perform the dial sequence. It will swap, as needed, from tone to pulse dialling and insert the pauses to allow a PABX dial-tone.

If you fail to make contact, and wish to redial, a single command '/A' can be sent from the keyboard (it is not preceded by an 'AT') and this is interpreted to mean 're-run the commands in the buffer'.

Modes and ranks

NOW WE ARE going to jump ahead and look at a typical 'asynchronous' modem – the type that you would most likely purchase at the present time for general home or business – and see how it handles the various data-rates and national modem standards. Many brands fit this model, but almost all of them use the same Rockwell or Mitel modem chips, but they would have their home-written command chip.

The set of commands that most modem manufacturers have used in the past has constantly needed to be updated, so each implementation of the AT-command set might be slightly different – even when the same modem chips are being used. When the manufacturers say that they comply with the standard Hayes AT-set, you've got to ask: 'At which level?' and 'With how many additions?' So take the following as a general overview of how the AT-command system works, and remember that your modem may be slightly different.

Let's assume that the modem model we are talking about would provide you with both Bell and CCITT standards, at data-rates of, say, 300 bps, 1200/75 bps, 1200 full-duplex, and perhaps 2400 bps, and that it has no physical switches.

I left you with a few un-answered questions last month:

□ When making a call, how does the modem select the correct data-rate from those available to it? There is no AT-command that says: 'Set this to 1200 full-duplex.'

□ How does the modem select the correct national standard – Bell or CCITT? There's no AT-command that specifically says 'Use CCITT standards.'

□ How does a modem make these choices when it answers a call and doesn't know which standard/data-rate the caller will be using? The caller could be using any of the wide range of mode possibilities.

The terminology gets pretty confusing when you begin to examine these questions, so I'm going to define two terms:

Mode – this is a unique combination of a data-rate, a choice of full- or half-duplex, and a national standard. So V.21 is a 'mode': it is the CCITT's full-duplex standard for 300 bps.

Rank – Like taxis in a rank, these are a selected group of modes that are made available for use. For reasons which will become clear (I hope!) only a few of the mode possibilities are available for use at any one time. If you count all the data-rates supported, the full- and half-duplex variations, and note that these are available in both CCITT and Bell, your modem chip might have a dozen 'modes' available. But at any one time, only one, two, or three might be 'active' (usable without making a rank-change) – and these constitute the 'rank'.

You won't find 'rank' in your manual – they'll just talk about 'modes' and 'ranges' which gets pretty confusing. By my count the manuals use 'mode' five different ways when refering just to the data-rate selection system, and 'range' at least two different ways. Modem manual writers believe that life wasn't meant to be easy.

In the Hayes command set, the variable 'B' defines the 'rank' selected for use. The B0-rank activates three CCITT 'modes' (V.21, V.22 and V.22bis) for 300 bps, 1200 bps, and 2400 bps full-duplex communications, and the B1-rank does the same for three Bell 'modes' (Bell 103, 212A and 2400).

In some modems these are the only two 'ranks' available, so the B-rank change is simply used to switch from CCITT standards to Bell standards, or vice-versa.

Later a whole lot of new 'ranks' were added because more choice was required (for the videotex V.23 standard, for a start), and so you sometimes need to change a modem's rank (the group of modes available for use) by sending it a B-command.

How does the modem select the correct national standard – Bell or CCITT?

For instance, if the command 'ATB2' ('Attention; set variable B to the value 2') is sent to our hypothetical modem it could only operate (sending and receiving a call) using three modes: V.21 (300 bps), V.22 (1200 full-duplex) and V.22bis (2400 bps).

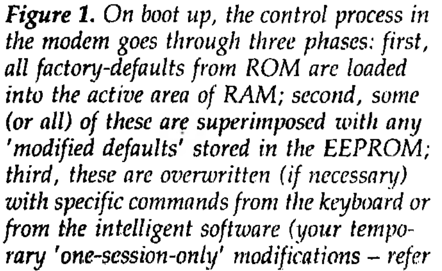


Figure 1. *On boot up, the control process in the modem goes through three phases: first, all factory-defaults from ROM are loaded into the active area of RAM; second, some (or all) of these are superimposed with any 'modified defaults' stored in the EEPROM; third, these are overwritten (if necessary) with specific commands from the keyboard or from the intelligent software (your temporary 'one-session-only' modifications – refer to the text).*

You will notice two things about this. Firstly, that these are all the same national standard (CCITT), and secondly that they all use a different data-rate. This last point is important.

Rule 1: *You can't have a B-rank which has two modes providing the same data-rate.* This is because there is no AT-command to set the data-rate, so the modem must make the decision itself – and it does this by judging the speed at which the communications software sends it data. If it receives an instruction (say, to dial a number) at a rate of 300 bps, then it will choose to use the V.22 mode (provided that this is available in the chosen B-rank). If the dialling instruction arrives at the modem at 1200 bps, then it will choose to use the V.22bis mode, if it is available.

So the B-rank selects what is active and available for use, and the data-rate of the next instruction decides which of those available will be used.

There is a potential conflict in only one case – that of the V.22 (1200 bps full-duplex mode) and the videotex V.23 (1200/75 bps hybrid half-duplex mode). Both share the use of a 1200 bps channel, and therefore you can never have both 'active' (available) within the one selected rank.

Rule 2: *You will always need to change B-ranks if you wish to swap from 1200 full-duplex communications to 1200/75 videotex communications. The modem can only ever be preset to provide one or the other.* This conflict is the basis of 90 per cent of the problems that people have with modems at the present – people either forget or don't understand that they've got to change ranks to use these two standards. (However, some software does make this change automatically.)

If the B2-rank is your modem's factory default, and you need to use videotex regularly (but rarely 1200 full-duplex) then set your modem's EEPROM default to the B0-rank with the command:'ATB0&W' ('Attention: Set the B-variable to 0 in RAM, then write this change to EEPROM'). If you want to use it temporarily only, drop the '&W'.

For the record (and these should all be in your manual) the B0-rank generally provides two or three modes depending on what data-rates are available. Our 'model' modem would have:

□ V.21 300 bps (the same as the B2-rank);
□ V.23 (the videotex 1200/75 mode); and
□ V.22bis 2400 bps (the same as the B2-rank).

In the B2-rank, our modem will have:

□ V.21 300 bps (the same as the B0-rank);
□ V.22 (the full-duplex 1200 bps mode); and
□ V.22bis 2400 bps (the same as the B0-rank).

In other words, these two ranks only differ because one provides the 1200 bps full-duplex mode, and the other provides the 1200/75 videotex mode. That's what the B-rank change is for. It is a very clumsy system – but unfortunately it is part ofthe Hayes standards.

Some modems have their factory-defaults set to the B0-rank, and others have them set to B2-rank. If you've bought your modem overseas, it may have the factory-default set to the B1-rank which provides the Bell standards: Bell 103 (300 bps), Bell 212A (1200 bps) and Bell 2400 (2400 bps). There is no American equivalent of the hybrid videotex standard.

If you've followed the above (and it is confusing) you should now see why you may often have problems with 1200 bps, while 300 bps works OK. The choice of the rank (made by the default, or by sending the modem a B-command) decides:

1) Whether Bell or CCITT standards are to be used;
2) Whether full-duplex or half-duplex;
3) Which data-rates will be available for selection; and
4) How many modes will be available in a rank (some have only one, others have three of more). Then the next command sent to the modem decides which (of the possible three) will be used. This is done by the simple procedure of the modem measuring the rate at which the software sends the command.

When making a call, how does the modem select the correct data-rate from those available to it?

The reason this causes so much trouble is that most people use the default B-rank setting (from the ROM or the EEPROM), and forget to change it when they go from a videotex system to one that requires 1200 bps full-duplex (or vice versa).

This problem is compounded by the fact that American communications software is widely used in Australia, and most US software doesn't provide a videotex selection anyway, so a modem with a default of B0 provides the videotex standard at the 1200 bps setting, but the software can't use it. Videotex wasn't part of the original Bell range of data-rate standards, so American software ignores the problem, and there's therefore nothing in the menu selections to remind you to change the B-rank.

I've just had a look at the Australian NetComm software to see what they do about this problem, and they have taken two steps. The first is that they force you to make a primary choice between accessing videotex systems, and 'normal' systems. When you enter the phone number for a videotex service, the software will automatically send to the modem a 'ATB0' command before it begins dialling to make sure that the B0-rank is available.

The second is that the non-videotex section of the software allows you to enter a B-rank command along with the number to be dialled. So, even if your software defaults to B0, and you need B2 every time you dial a certain number for a 1200 full-duplex connection, the B-rank change process can be automated, so you won't forget.

This is a very sensible approach, but unfortunately most software doesn't have this built in. You are expected to enter Terminal Mode (where you can send instructions manually to any modem) and send the rank-change command before you use the software.

This is only a problem for me when I want to use 1200 bps – so another solution is to avoid using this data rate entirely for normal non-videotex use. That's what I do, and it works well, so I'm going to make it into a rule.

***Rule 3:** Leave the B0-rank as the default to provide videotex, but for PC-to-PC or PC-to-network communications only ever use 300 bps or 2400 bps. Never use 1200 bps unless for videotex.*

Answering modes

THE AUTOMATIC answering process is a bit more complicated. My software has a selection 'Wait for Call' – but there isn't any specific AT command that tells a modem to wait; the modem is just automatically in the answering mode unless instructed to dial.

What the 'Wait for Call' menu selection does, is to send an 'ATZ' command followed by an 'ATS0=1' command to the modem at the data-rate set in software. This performs several important functions.

First, the Z command sets the modem back to the 'default' condition (those stored in the EEPROM). This is the equivalent of switching off the modem and switching it on again – it clears RAM and restores ROM+EEPROM changes. They've included this because you don't want temporary change made for one session, necessarily carrying over into the next – so this sets everything back to a known starting point.

Second, the 'ATS0=1' instruction is simply to reset the S0-register (which controls the number of rings before the modem will pick up the line) to 1. Sometimes people play around and accidentally alter these numbers, and if the S0-register is set to 0, the modem will never answer.

Third, the rate at which the 'ATZ' command arrives at the modem establishes the mode that the modem is now set to expect (chosen from those available in the preset B-rank). This last point is important.

Only the more expensive modem models can 'Auto-range' from one mode to another until they find a match: most modems will sit waiting for a call in the one chosen mode. When the call arrives they will either accept or reject connection on this basis alone: if it doesn't match the mode expected, you'll get a 'NO CARRIER' response.

So with any of the cheaper modems (under $1000) you need to know what data-rate the calling party intends to use before you boot up your modem.

If you are setting your modem manually through Terminal mode, remember that the modem will wait for a single mode established by a) the B-rank; and b) the data-rate of the last instruction it received. The trap here with some software, is that there is no mechanism to automatically update this mode selection if you make a change after issuing a 'Wait for Call' instruction.

Say you are waiting for a call. Suddenly you realise that you've set the wrong data-rate, then you must:
1) change it; *2)* enter the Terminal Mode; and *3)* type 'AT' and get an 'OK' back in reply.

Only now will the modem have switched to the new data-rate, because it always sets itself to the rate of the last instruction, and not all software automatically sends an instruction in these circumstances. The only way to find out whether your software does an automatic update in these circumstances is to run a test.

Other parameters

YOUR MANUAL WILL have a lot of different control variables which are generally best left at the factory-default settings. However you can change them temporarily, or semi-permanently, as you wish. You can always restore the modem to the factory defaults (or to your modified defaults) so it doesn't hurt to play around with them.

The store and restore commands in the Hayes AT-set are – ***AT&W*** which writes any changed parameters in RAM into the EEPROM (or non-volatile memory);
ATZ command which restores the modem's active RAM parameters to its 'modified defaults' (the factory defaults + any overwritten EEPROM changes); and
AT&F which temporarily re-writes the factory defaults from ROM into active RAM.

If you want to write these ROM factory defaults into the EEPROM (in effect, wiping out the old changes), you've got to issue the 'AT&F&W' command. □

Alterings

TO CLARIFY these 'read-write changes to various types of memory, let's alter the number of rings the modem will wait before answering, and follow it through various stages:

□ The factory default in the ROM's S0-register (an 'array' variable) holds the parameter value of 1. This means that when answering, the modem will pick up the phone on the first ring.

□ On boot-up this value is loaded from ROM into the S0-register space in the active part of RAM, and this becomes the parameter that the modem's microprocessor checks when deciding when to pick up the phone.

□ We modify this (for one session only) to 'two rings' by sending the instruction 'ATS0=2'. We must use this '=' sign with register changes because a number is part of the variable name.

□ If we wish to make this change semi-permanent (so it is the effective default on every future occasion), we could include this write instruction in the original command line as 'ATS0=2&W'. The change is now both in RAM (temporarily) and in EEPROM (semi-permanently).

□ Suppose we now decide that, for this session only, we will set it to ring three times. We can now issue the instruction 'ATS0=3' and the parameter 3 will be stored in RAM and become the dominant value for this session. We now have the parameter of 1 at the ROM factory-default level, 2 at the semi-permanent 'modified default' EEPROM level, and 3 at the active RAM level, and 3 is 'dominant'.

□ To restore the modem to its working-default condition we issue the 'ATZ' (reset command). This causes all modified parameter values in the EEPROM to overwrite those currently active in RAM.

□ If we wish to restore the factory default values from ROM for some reason, we would issue the command 'AT&F'. This makes no changes to the values in the EEPROM, only to the values in active RAM.

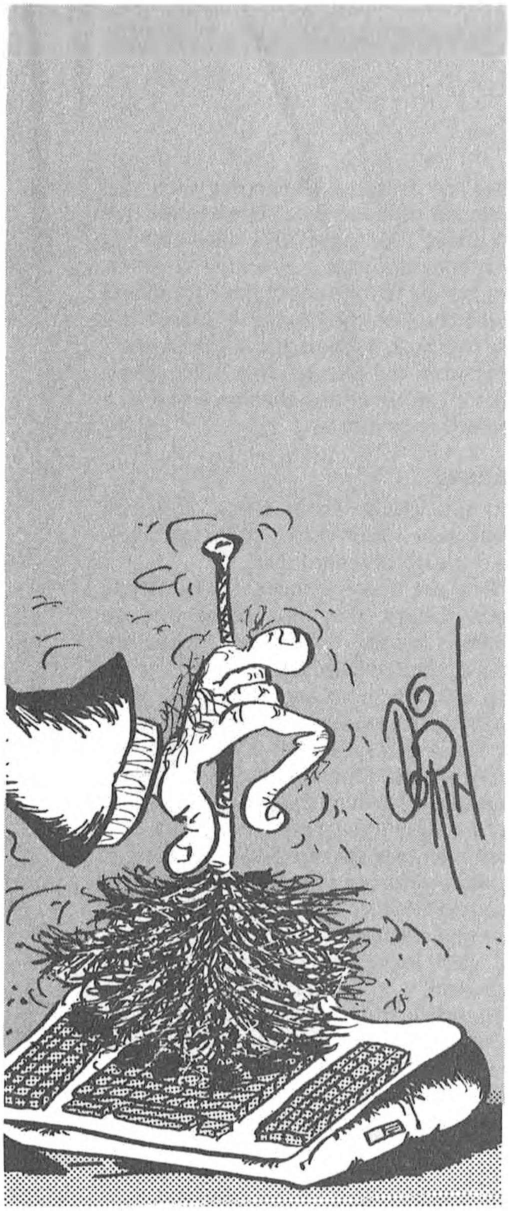
□ If we now needed to wipe out any changes in the EEPROM, we would command 'AT&W' to write all ROM=RAM changes into the EEPROM once again.

PC MAINTENANCE

IF IT'S SIMPLE, IT GETS DONE!

As Bruce Iliff tells, your investment in PCs, software and accessories needs protective maintenance – and don't forget the most basic of computer maintenance: backups.

YOU'VE JUST bought a new personal computer. Now plug it in and start thumping the keyboard? Wrong! Like any machine, a computer needs a certain amount of maintenance. Naturally, everyone can't be a computer technician, but there are certain things you can do to keep your machine in good condition.



The integral component of most machines is the hard disk. If this fails then the entire machine can be virtually useless, regardless of whether it is a '386 running at 25MHz, or a simple XT compatible There is little a user can do to care for the hard disk as it is completely sealed to keep out dust and dirt.

One thing you can do though, is to park the disk after every session. The read/write heads are micro-inches above the surface of the disk. Any bump can cause the heads to score across the surface, effectively ruining the disk. Parking the disk keeps these heads away from the data area, so if something happens while the machine is off, the disk won't get damaged. An example of what can happen is the computer desk collapsing – it's happened to me. As an outlandish example, if the office or home is burning down and the hard disk is parked, it can be picked up while making a wild flee from the burning rubble! Some machines automatically park the disk when the power is off, while others require a utility, HDSIT or similar, to be run before switching off.

As the only moving parts in a personal computer, hard disks are prone to failure If a hard disk has lasted five to seven years, it will be near the end of its life. After this, it would be wise to replace it, especially if it contains a lot of expensive data. And, with the rapidly evolving market, it's a good justification to update the entire system!

Naturally, keep a desk-bound machine on a desk. Laptops use a different type of disk drive which gives them rugged portability.

Floppy disks

WE ALL KNOW how to care for floppy disks – don't bend them, spear them with a chair leg, or staple labels to them Storage is one way a lot of data gets lost. I heard about one fellow who kept his disks under his telephone. He always wondered why they were getting corrupted, and eventually, he was told that the magnet in the telephone bell scrambled the data.

If you have a lot of expensive data stored on floppies (backups for example), it might be worth buying a fire-proof safe for storage, as it could save a lot of heartbreak.

Remember that small piece of cardboard in the floppy drive when you bought the machine? Well, that is to put in the drive to stop the twin heads bouncing together when you move the machine. It's not a complementary partition for your disk box!

Everything works well until Murphy comes along, or, until he spills coffee over your keyboard. If this happens, expect to buy a new one. If any liquid gets inside the main unit, turn it off immediately and take it to a repair technician, and don't forget your cheque book!

Keep Murphy and his damn cigarettes away from the machine. Smoke quickly infiltrates computers putting a coating on the pins of the ICs, the heads of floppy drives, and even on the print head of a dot matrix printer.

If the machine 'hangs up' or 'freezes' – as happens a lot with Murphy – and the big red switch is required to restart it, always wait at least five seconds before turning it back on. When a machine is turned off, some voltages take a while to dissipate. If they are still there when the power returns, spikes and other problems occur that could eventually damage certain components.

Hands up who's unplugged a peripheral cable while the machines at either end are turned on? We can get away with this a thousand times, then one day along will come Murphy, wanting to change printers. Problems will occur because you're effectively breaking an electric circuit. Voltage spikes can be generated because it is virtually impossible to pull the plug so all circuits break at once. This minor arcing and sparking can result in either machine having an internal rupture. If you have two printers, or two serial devices – a mouse and a modem – either put in another port or use a switch box which breaks all the circuits at once. These are quite cheap considering the problems they solve, and the cost of the equipment at either end.

Cleaning

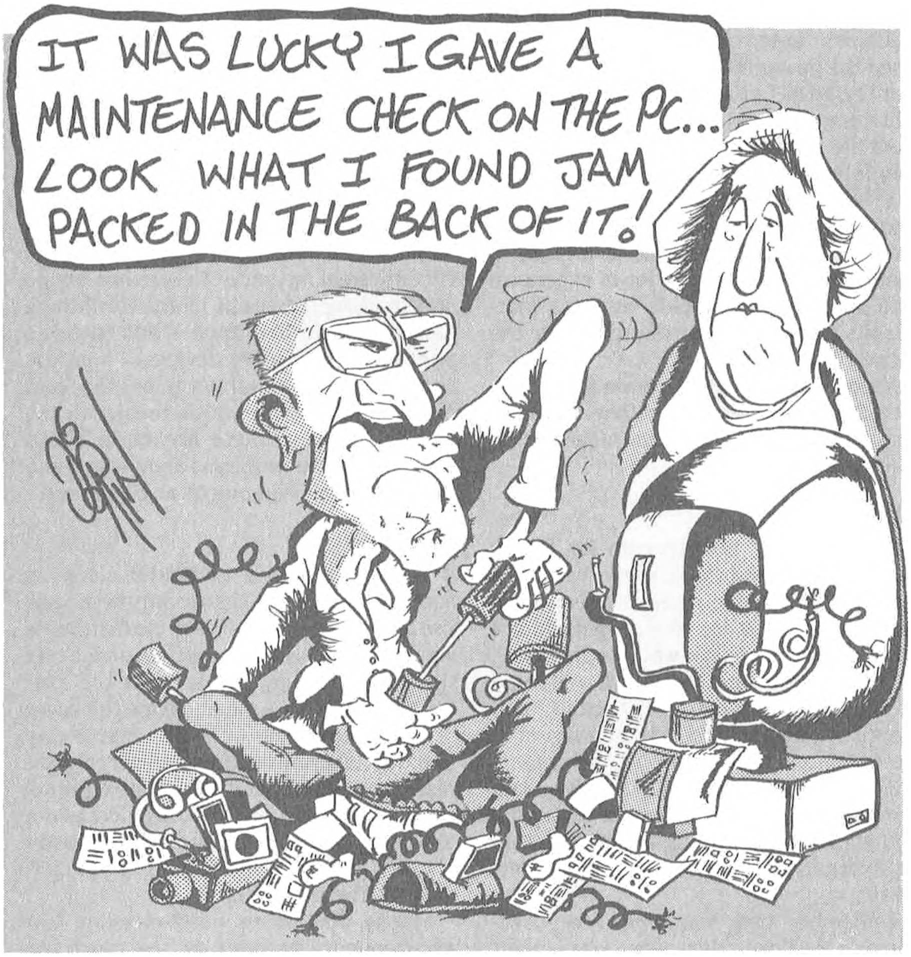
IT PAYS TO keep a computer clean as nothing looks worse than a dirty keyboard. Use a domestic type liquid cleaner, making sure you turn the unit off and never spray the cleaning solution on the keyboard – put it on a rag and wipe the keys.

Always keep your work area free of dust and dirt. Like smoke, dust can get into the main unit. Keep your dot matrix printer clean of that fine paper dust by occasionally using a vacuum cleaner with a narrow nozzle, as paper dust will eventually smudge the print head.

Floppy disk drives need cleaning too, and frequency depends on how much you use them With a hard disk, the floppies don't get as much use, but with a twin floppy system, the heads can quickly accumulate dirt. A lot of this comes from the coating of the disks, and the only way to clean the heads is with a proper disk cleaning kit. But, make sure you follow the directions carefully or you could end up with more problems than you started with.

Software is what makes a PC run. It's no use having the fastest machine on the market with masses of storage and the clearest screen if you don't maintain your software in some order.

Throughout this article, I've talked about 'expensive data'. In many business situations, data is worth more than the machine. If a computer dies, it can be simply replaced, but if the data is lost, it's gone forever. If you have a lot of data accumulated over the years, work out how long it would take to punch in that information. A simple mailing list of a thousand names could take a typist an entire day, and a ten thousand item inventory could take a week, that is, if the data existed someplace beside the computer.


IT WAS LUCKY I GAVE A MAINTENANCE CHECK ON THE PC... LOOK WHAT I FOUND JAM PACKED IN THE BACK OF IT!

And, don't forget the time the machine will be without that mailing list or the orders that aren't being placed because the inventory doesn't exist. This is all money, so data is expensive!

A backup ritual is extremely important. There are infinite variations – using piles of disks, a streaming tape, or a dedicated machine just for a backup, even a combination. Whatever you chose, make sure it is done regularly. Once a month, every week, or whatever your particular situation demands – do it! (Too help you get started, the box item 'No excuses' describes a wide range of backup software and who to contact for more information.)

If using disks or tapes, a good system is to take a 'last resort' copy and store it away from the machine. You can store it in the local bank if the data is expensive, or take it home with you. Refresh it every six or twelve months, and then start a rotating system with three sets of backups, one taken every week. This is called a Grandfather-Father-Son backup. Each week you write over the Grandfather copy, which then becomes the Son, and the Father becomes the new Grandfather. In this way, you will always have three sets of data, the oldest being three weeks. Then if the data on the hard disk gets corrupted and takes a week to notice, you might have backed-up the corrupt data last week, but you still have two backups containing good data.

Naturally, no backup system is infallible, and Murphy sees to that. So it is wise to regularly take hard copies of all data and store it off the premises. If Murphy does play havoc with your backups, it might only be a case of inputting the data again, and this could be better than starting with nothing.

For those on a budget, floppy disks are the most economical. You can Dos' Backup command, or get hold of a utility package to make the job easier. If you have a menu system, this is the place for any backup commands. It makes the job simpler, and if it's simple, it gets done!

Fastback is one of the better utilities for an IBM compatible, and the latest version costs a little under $200. It has a wide range of features depending on your particular use, and data compression can put about 500K on a 360K disk, and can verify the data after it is written. It generates history reports about the backup and stores them on the hard disk. Even if you lose this history report, it can generate another from the backup disks themselves. Macros can be written then accessed by a single Dos command.

Viruses

WE ALL KNOW about these. They can wreak more havoc than a spilt cup of coffee or death of a hard disk.

They are mainly propagated by floppy disks, though they can come through bulletin boards. Nearly all types, or strains, change COM and EXE files or attach onto the boot sector of disks. With the former, the virus is only initiated when the file is executed, then it wakes up and can wipe data, format hard disks and start propagating to other executable files. One way to stop the catastrophic affect of a virus is to only backup data files. There is no way a virus can attach onto these – yet! If you do find a virus on your hard disk, it's a simple matter of re-formatting, re-loading your legitimate software, and then your data.

Naturally, this only works if you are using software registered for your use. If you are using pirated software, the virus could be in the pirated copy. The only way around this is to buy legitimate software.

Be aware of what is going into the floppy drive or through a modem. Always buy software from legitimate suppliers and be wary of anything from Murphy down the street with the part-time public domain library. Get hold of a 'virus buster' or similar type of program (there are a few coming out on the market). Run this regularly to check the integrity of your system, and put it on the menu to ensure it gets done. Always be alert to what your machine is doing. If the floppy drive is accessed for no reason or if the computer is running slower, it would be worth checking for a virus.

A common way a virus spreads is through a business machine with a laser printer attached. Employees with home computers use the printer for the final copy of their personal letters. All it takes is one infected disk and it quickly spreads to numerous computers. If the business machine is cured, it can be infected again when Murphy comes along to print the final draft of a letter to Aunt Agnes.

Housekeeping on a computer means

No excuses

THERE IS NO excuse for not protecting the data on your hard and floppy disks The hardware needs its own protection and maintenance, but, after all, hardware is replaceable, but much of your data won't be, or it will require a concerted investment in time and lost work to recover it.

The simplest form of backing up data is simply to make a copy of everything. That's fine as long as everything fits on one floppy – when it doesn't, the next step is to use one of the public domain compression programs After that, backing up can start to be messy and time consuming – which means it will be done less and less. Like any 'security' procedure it needs to be simple, to be effective.

Even MS-Dos can be used to simplify backing up: check your Dos manual on the Backup and Restore commands. But, using Dos will only remain an effective method while the volume of data to be backed up doesn't occupy more than a few floppy disks. After that, the time involved and the inconvenience of swapping floppies starts to detract from the effectiveness.

For a system with only a single user, swapping floppies might still be the answer – it's a matter of balancing priorities. But, for larger systems, even with only two PCs, a backup procedure automated with software is the only method that will work consistently (it's a corollary of Murphy's Law, that the need for backups is directly proportional to the time since the last one was done). You will be entrusting the security of your data – and all of the investment that implies – to any backup software you undertake to use, so choose it carefully.

Try at least three different programs before deciding to ensure you feel comfortable with the way it works. Once it's installed, make sure that all operators understand why the software is there and appoint one user to oversee backup procedures.

Our survey of backup software showed most packages fall between $100 and $200 – that's a small investment to protect a lot of work. Some of the current offerings on the market include –

Back-it v3.1: $185; from McQuarrie Management Services, ph: (02) 958 2945, fax: (02) 958 8671.

DS Backup+ v2: $185; from Computer Equipment News (formerly Software Wholesalers), ph. (02) 957 6686, fax: (02) 957 2189.

Fastback Plus v2: $329; from Programs Plus, ph: (08) 326 1313, fax: (08) 326 1644

Intelligent Backup: $242; from Imagineering, ph: (02) 697 8666, fax: (02) 697 8650.

PC-Fullbak+: $179, from Portfolio, ph: (02) 487 2700, fax. 489 1265

PC Tools Deluxe v5.5: $149; from Vaporware, ph: (02) 725 3700, fax: (02) 604 1983.

Sitback v2: $169; from Logo Distribution, ph: (02) 819 6811, fax. (02) 819 6930.

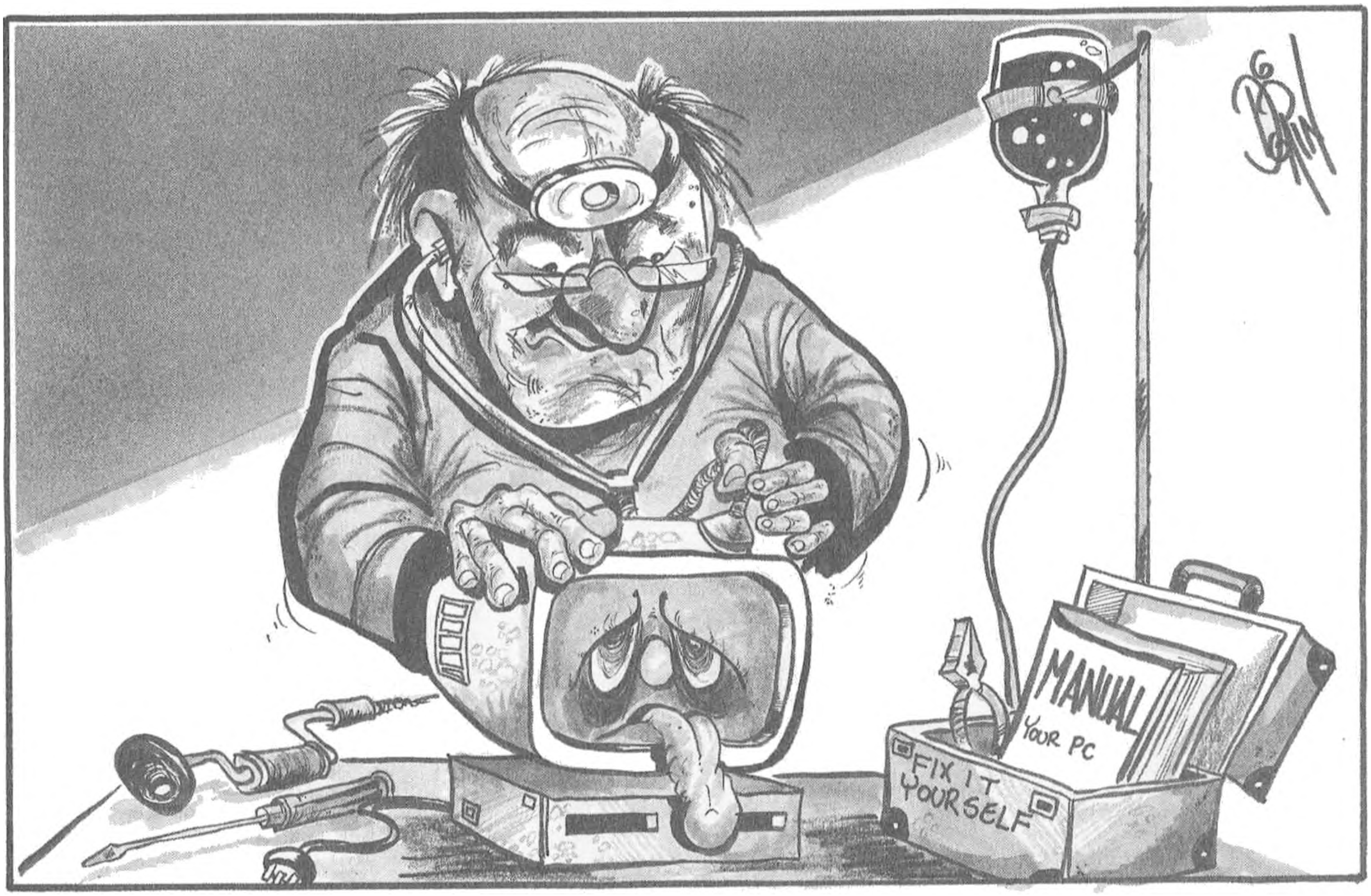
Generally, you will find that the features offered with each package are reflected in the price. To give you an idea of what's available, it would be a worthwhile exercise to seek information on two or three different products listed above.

keeping things in a logical order. Arrange directories to keep track of files, regularly clean out unwanted files, and set up a menu system for easy and quick access.

After a while, files on a hard disk get fragmented and are stored all over the disk. This can reduce disk access time and cause more wear on the heads. The way around this is to reformat the disk, or use a utility like Norton to tidy it. To keep the disk in the best working condition, it is worth formatting every six months. The magnetic coating on the disk naturally deteriorates and reformatting revitalises it.

All these maintenance jobs can keep your computer in good working order to help in those mundane tasks for what they were designed. But always remember that no matter how much care and attention you take, Murphy is always there ready to strike! □

FIX IT YOURSELF!



PART 1

Fixing your own computer problems can be a rewarding experience, in more ways than one. All it takes is a little understanding and logic as Robert O'Rourke explains in the first of a three-part series.

TO SAY THAT a computer is a complex electronic device would be stating the obvious, at the very least. This complexity causes many people to shy away from attempting to repair even the most simple problem, for fear of making things worse, and adding to an already unwanted repair bill. That's a natural enough fear for someone who knows nothing about the inner workings of a PC, but as we will see in this three-part series, there are many PC problems which can be easily fixed by someone with only a little knowledge about what's under the 'hood'.

Consider the ubiquitous automobile. When a problem arises with the family sedan, the owner quickly shows allegiance to one of two religious orders. One group runs for the *Yellow Pages*, and looks for the nearest specialist. This is the simpler approach, but for a vehicle out of warranty, it always corresponds to a rather hefty dent in the bank balance.

, Members of the other group, however, reach for the workshop manual on the bookshelf, lift the bonnet, and proceed to track down the problem. This method is the more time-consuming of the two, but there are at least two advantages. The first is that you will almost invariably save a lot of money – you still need to buy any parts that are required, but the labour is yours. Also, you might actually learn something about what you are fixing, which, at the very least, can stimulate some disused neurons into action.

So, the next time a computer under your care packs it in, think awhile before sending it off to the local PC garage. Often the problem can be fixed with a little work on your part and, in many cases, this can be completed in a much shorter time span than the typical turn-around from a computer repair shop.

Fixing anything of the complexity of a computer requires a logical approach to the problem. An understanding of how the computer works at an electronic level is certainly an advantage, but all that is really needed is a knowledge of what components other parts of the system depend upon for proper functioning.

In many ways, diagnosing a computer problem is little different to pin-pointing a medical problem. The doctor knows the symptoms of common illnesses and, in many cases, a diagnosis can be arrived at by analysing the symptoms of the patient – so too for the computer.

However, just as in medicine, it is also possible to make an incorrect diagnosis and, in both disciplines, treatment based on such a diagnosis can do more harm than good. Therefore, it is important to collect as much information about the problem before you jump to any conclusions. If you have made a diagnosis, and new information comes to light, then you must make sure that this new information is consistent with your initial theory. If not, then either the original diagnosis is incorrect, or based on false data, or the new information is itself in error.

One thing which comes in very handy when fixing a computer, is a computer. No, not the broken one, but a known good machine, preferably with a similar configuration to the sick one. This allows components to be swapped between the broken machine and the good one, in the hope of finding the fault – at least as far as figuring out which board is at fault, which is pretty much as far as we will be going here.

A final word before starting out on our adventure. Never connect or disconnect anything in the computer with the power turned on – always switch the power off, and wait a couple of seconds for any remaining charge in the system's capacitors to discharge. This is not for your safety, but that of the computer. With few exceptions (such as inside the power supply, and around the power switch), the voltages in your PC are no higher than those encountered in a car electrical system.

However, plugging devices in while the power is applied to the machine is courting disaster. The components in the computer are very fussy about the way in which power is applied to them, and having power connected to some pins and not others can cause irreparable damage to the machine. Presumably, you already have enough trouble with your computer (why else would you be reading this), without creating more problems.

Initial diagnosis

IT IS A FACT of life that the weak links in computer systems are the mechanical components. This not only includes such obvious parts of the system as disk drives, but also many less complex (but just as critical) things as switches, connectors on cables, and the edge connectors on the motherboard, into which the expansion cards plug.

Connectors generally fail where the pins in the plug make physical contact with the wires. Often, wriggling the cable at the point where it enters the connector can make the connection come good for long enough to determine that you are in fact looking at the cause of the problem.

If the machine is completely dead – that is, no sign of any activity at all when the power switch is turned on, then the most probable cause is a power supply problem of some sort. The first thing to determine is whether mains power is actually reaching the computer or not. Obtain another power cord, and substitute it for the one on the computer, and try it again. This shouldn't be too difficult, as the type of power cable used on computers is quite common now for all manner of appliances.

> *There are many PC problems which can be easily fixed by someone with only a little knowledge about what's under the 'hood'.*

If it is the power cord which is at fault, throw it out, and buy a new one. Don't try to repair it unless you are a qualified electrician. Too many lives are lost each year due to incorrectly-wired power plugs now, without you adding to the statistic.

If the power cord is OK, then the problem lies inside the computer somewhere, and it is time to roll up the sleeves and remove the lid. Inside the case, usually in the right-hand rear corner, is a large silver box. This is the power supply for the entire system, whose job it is to convert the local mains voltage (240 volts in Australia and New Zealand) to the much lower voltages required by the components in the computer. It also serves to stabilise the supply, to account for any voltage fluctuations in the mains supply.

There are two main supply voltages in PC compatibles – the 5 volt line (abbreviated 5V), and the 12V supply. The former supplies by far the bulk of power to the computer system, powering all the chips on the motherboard and expansion cards, and also the chips built into the disk drives. The 12V rail powers all the motors in the computer – the spindle motors for hard and floppy drives, and also the motor for the cooling fan.

There is also a –12V (negative 12 volts) rail, which is usually only used, in conjunction with the 12V rail, to power the line driver chips for any serial ports in the system. If this line dies, you will lose the serial ports, but rarely anything else, so we won't worry about it for the moment.

If the 5V line has died, you'll soon know it – the fan and hard disk motors might spin up, but very little else. If it's the 12V rail, then the computer will probably start up, but components like disk drives which rely on this rail for motors won't work. This will be reported as drive failure errors during the power-on self test.

So what to do with a bung power supply? Either replace it, or get it fixed. Unless you are familiar with the intricacies of switchmode power supplies, or know somebody who is, then the former is definitely the way to go. The power supply case contains potentially lethal voltages, so unless you know exactly what you are doing, leave it alone.

Just because you have a power supply problem, it does not necessarily mean that the power supply unit itself is at fault – it could just be overworked. The power supply has a limited capacity – not unlike the power circuits supplying your home or office power outlets. Put too much load on the power supply, and something has to give. In a mains installation, an overload will blow a fuse or pop a circuit-breaker, causing all power to be cut off until the problem is rectified.

The power supplies in computers are somewhat more intelligent – they shut down when they are overloaded. Resetting the power supply is a simple process involving turning the machine off, removing the cause of the problem, and turning it back on.

The original PC didn't have much power supply capacity at all – 63 Watts was all it could provide. This was enough to feed the motherboard and one or two floppy drives, whose motors never ran simultaneously, and a video card. But if you added a hard disk and controller, or a memory expansion board, then you would likely run into power supply problems. Fortunately, most PC clones are in fact clones of the XT, with its beefier power supply (130W), and some clones had even larger supplies still. Third-party power supplies are also available for those with old PCs, who want to add a hard disk or more expansion cards.

The only time you are likely to encounter power supply overload problems nowadays is when you are really cramming a lot into a single box, for applications such as file servers with many hard disks and memory boards. In these cases,

power supplies can be had which fit up to 300W into a standard-sized enclosure.

The power supply may not be the only cause of a total failure to come to life. This could be because there is an abnormal load on some critical line on the processor bus, which prevents it from communicating with the rest of the computer. Should this seem to be the case, remove all expansion boards from the system, including any I/O cards, but leave the video controller in place.

If you can then power the machine up, then the problem lies on one of those boards, and it is a simple case of plugging them in one by one, until you find the faulty one. Of course, if the system still doesn't work, you still have the video board and the motherboard to eliminate. First try unplugging the video board, and powering up the system. If the motherboard is working properly, the speaker should beep a couple of times, which indicates a video failure in the BIOS self tests. Since there is no video adapter through which to deliver any error messages, the BIOS resorts to the only other standard output device on the PC – the speaker.

If the speaker remains silent during this phase of the test, then you can be pretty well assured that the problem lies on the motherboard. The integrated circuits on the motherboard will often be mounted in sockets, to facilitate the process of replacing them. These sockets, like any others, are prone to mechanical failure and can also work themselves loose.

Preventative maintenance

PREVENTION, TO QUOTE the well-worn cliche, is better than cure, and this is as applicable to computer problems as anything else. Large computer installations have preventative maintenance performed on a regular basis, but users of PCs tend to take them for granted, and only worry about the health of the machine when something is obviously amiss.

Some preventative steps do not even involve removing the case of the computer, although any good program will necessitate this once every few months or so. Without doubt, the biggest enemy of computers is dust, and other airborne particles, and this is one of the reasons that mainframes are housed in such well-controlled environments.

Unfortunately, most PC manufacturers do not take any steps to prevent the ingress of dust at all. Airflow is generally provided by the fan mounted in the power supply, but the problem is that this fan sucks the air out of the case of the computer. The air is free to enter the computer through any opening in the case. Some are designed as air inlets – one is usually mounted in the diagonally opposite corner to the fan, in front of the expansion cards. Other air inlets are accidental – like the openings in the front of floppy drives.

It's cheap insurance, even if it does wear the drive heads somewhat faster than would otherwise be the case.

None of these inlets has any filtering on it and, if they did, their effectiveness would be reduced because the air would then find easier paths into the computer. Life would be far simpler if the fan *blew* air into the case, and filtered it at the same time. Perhaps the engineers thought users would neglect to clean the filters, resulting in restricted airflow and overheating.

Whatever the reason, we are now stuck with computers that fill up very rapidly with dust. Once every two or three months, you should remove the case of the computer and, using a small vacuum cleaner, remove any build-ups of dust which are visible. Take care that you don't damage the circuit boards while doing this, or the whole operation could become counter-productive. A small fine-bristled brush is useful to dislodge dust from components, making it easier to pick up with the cleaner.

If the interior of the computer is relatively free of dust, you could extend the interval between check-ups. Conversely, if the system is totally clogged up with dust, then it would be prudent to clean it out more often.

Disk drives also accumulate dust, both inhaled through their front openings, and deposited from the surfaces of disks that are inserted in the drives. A good non-abrasive head cleaner will remove the build-ups on the drive heads, where it interferes with the signal transfer between the heads and the magnetic coating of the disk.

Whether drive cleaning should be performed as a preventative measure or not is subject to some debate – the same debate that has raged for years in relation to audio and video tape heads. The argument against unnecessary cleaning revolves around the abrasiveness of the cleaning process. While head cleaners aren't as abrasive as they used to be, the removal of abrasive particles is bound to cause some wear of the heads.

However, letting the build-up progress too far is also potentially damaging – not to the drive itself, but to your data. Certainly a potentially more expensive event, especially if you are not aware of it at the time.

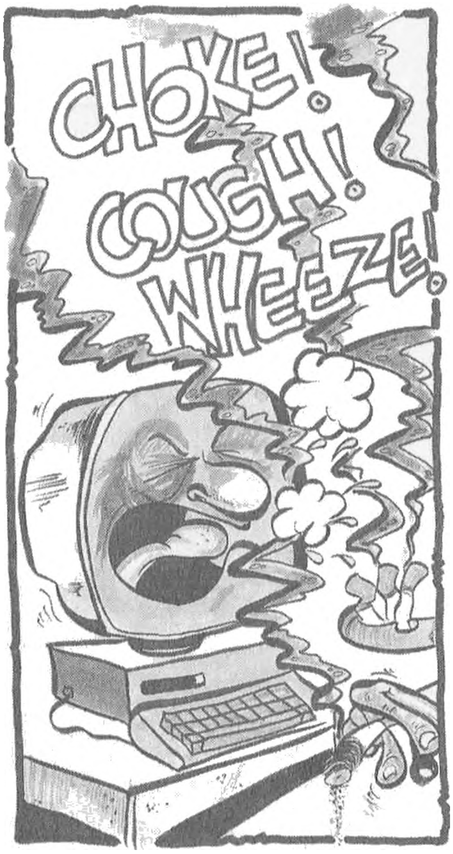
Although dirty heads usually manifest themselves when reading disks, the build-up will also effect disk writes, so the signal actually written in the surface of the disk is weaker than it should be. If you have a disk that you can't read because of dirty heads, you can usually clean the heads, and then proceed to read the disk. However, if you can't read the disk because the heads were dirty when the disk was written, then there's usually nothing you can do about it. If the unreadable disk constitutes part of your last backup, then you've got real problems.

That said, I have to admit that I don't usually remember to clean the drive heads on my machines until I can't read a disk that someone has given me. But, if I am writing to a disk that is particularly important, such as doing a backup, or copying an article to floppy, then I will clean the heads before performing the operation. It's cheap insurance, even if it does wear the drive heads somewhat faster than would otherwise be the case.

Just don't smoke

I NEVER CEASE to be amazed at the number of people who smoke near their computer. I guess that the assumption is that since the computer is not housed in a climate-controlled room, it is impervious to atmospheric pollutants such as cigarette smoke. The big hazard with smoke, as opposed to ordinary dust, is that it contains tar, which quickly coats every exposed part of the computer, making it all that much harder to remove dust which sticks to it later on.

Remember that dust not only interferes with mechanical components like disk drives and edge-connectors, but it also effects the operation of many of the chips, by reducing their ability to keep themselves cool. The dust builds up on their surface, insulating them from the cooling air flow provided by the fan. A thin coating of tar helps make sure that any airborne dust sticks to the chips, making this insulating coat build up rather quickly, and also making it difficult to remove.

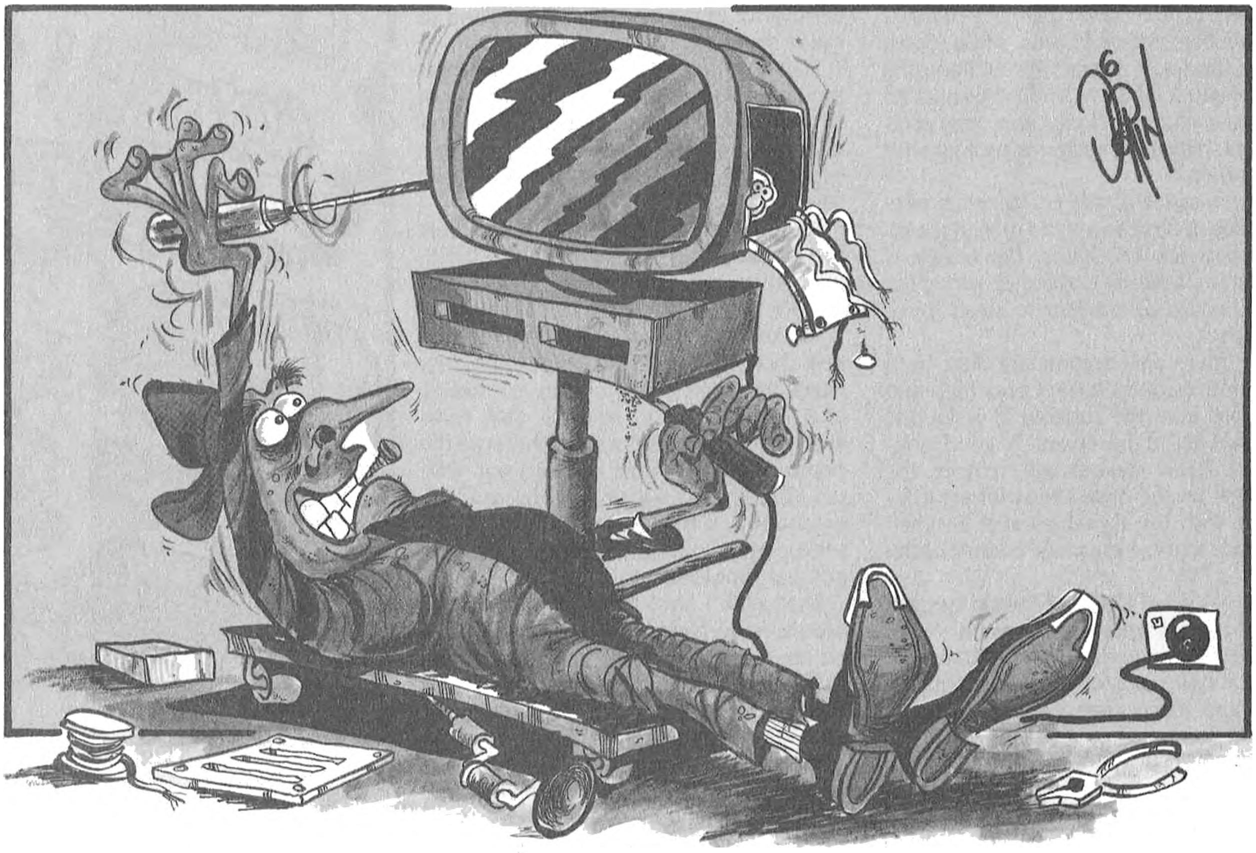


The connectors joining the various components of the system together should also be checked. Make sure that the mating plugs are pressed fully home, and that the cables attached to them are not placed under any undue stress, which could potentially cause problems in the future.

External connections, such as to the monitor, keyboard, and other peripherals, should also be checked when plugging the computer back together. D-type connectors (so called because they look like the letter 'D') have locking screws, either with standard slotted heads, or with thumb screws, to stop them falling out of their receptacles. Making sure these are secure when the plugs are inserted, will eliminate them as a source of future trouble. Make sure that cables are kept away from wandering feet and chair legs – it doesn't take much force to rip the wires from the pins in one of these plugs.

Next month, we'll look at curing some of the most common problems with computers – those involving disk drives. Of course, as I said before, prevention is preferable, but when something *does* go wrong, there is a lot that can be fixed without taking the machine to the computer hospital. □

FIX IT YOURSELF!



PART 2

In part 1, Robert O'Rourke discussed simple fault diagnosis and preventative maintenance. This month he looks at the most problematic components of all – disk drives.

OF ALL THE major components in a PC, the only ones containing moving parts are the hard and floppy disk drives. The combination of normal wear and tear on such moving parts, added to ever-present foreign objects fouling the workings of drives, make disk drives the most common cause of computer problems.

With that in mind, if you encounter a computer with what appears to be a drive problem, it is a good idea to analyse the entire disk sub-system, before jumping to any conclusions about the drive itself. A typical PC has two separate disk sub-systems – for both floppy and hard disks.

The drive sub-system consists of a controller and one or more drives. The fact that most AT machines combine the hard and floppy controllers on the same circuit board, does not alter this fact for the purposes of fault analysis, although it means that if there is a fault in either controller, you will probably have to replace both.

Of course, you will want to determine whether the problem is with the drive itself, or with the controller. The only reliable way to do this is substitution – plugging known-good components into the computer in place of the suspect ones, or plugging the suspect ones into a good computer.

If you have a two-drive system, and neither of them work, then the problem is almost certainly in the controller. On the other hand, if only one works, then it could be either the drive or the controller. Try swapping the connections to the drives (and change the appropriate DIP switches or CMOS settings if they are different types), and see if the fault follows the physical drive, or the logical drive name. If you still get the errors on the same drive letter, which is now associated with a completely different drive mechanism, then the problem is obviously not with the drive. Conversely, if the problem follows the drive when you change them over, then you've cleared the controller of suspicion.

If you come to the conclusion that there is a fault in either the floppy or hard disk controller, then it will have to be replaced (unless you want to repair it, in which case I'll leave you to it). The question now arises as to which controller to buy to replace it.

Obviously, the replacement must be compatible with both the rest of the system, and, in particular, the drive which it will be called upon to control. SCSI and ESDI drives will only work with a matching controller, so there is no room for change here. However, you may want to consider getting a higher performance controller, such as one with an on-board cache. I won't go into details about this here, but the point is that if you think you might want such a device in the near future, it could work out cheaper if you get it now.

If the hard drive is an IDE type, then about the only choice that you will have to make is whether to get one with a floppy controller on the same board, or just a new controller. If the faulty controller controlled floppy drives as well, then you obviously won't want to lose your floppy drives in the process. However, you may be able to get away with just getting a hard disk controller, if the old card has a jumper to allow it to be disabled. Many cards have such a jumper, so that you can leave the old controller in place to run the floppy drives, and disable its hard disk controller, using the replacement part to take over the job.

This approach obviously takes up an extra slot on the motherboard, but if you have a few spare, and don't have any plans which might require all of them in the foreseeable future, then this approach will save you a few dollars. It also applies to other combined hard/floppy controllers, provided that all-important jumper is present to disable the broken part of the card. Just because the controller is broken, doesn't mean that it won't *try* to work, and most likely interfere with the new one.

On the other hand, if you have a faulty hard or floppy disk controller, and they are on separate boards, then you may want to take the opportunity to combine their functions, in order to free another slot. In this case, simply remove both cards, and connect the drives to the newly-installed combo card. You can also get these cards with a parallel port, two serial ports and a games interface – all on the same card. So if you need (or want) to add any other ports, this could be a way to do it without taking up any more slots.

There are some important differences between hard disk controllers suitable for XT and AT machines, which you should be aware of before shelling out money for a new card. The most obvious of these is the fact that XTs use an 8-bit controller, while ATs use a 16-bit one. This difference manifests itself in the size of the edge connector on the lower side of the board – the XT card has a single 62-way connector, while a controller for an AT has an additional 36-way one adjacent to it.

The number of bits refers to the amount of information which the controller can transfer over the bus in a single bus cycle – so an AT controller can transfer data twice as fast as one for an XT. But you can't put a 16-bit hard disk controller into an XT, since the XT only has an 8-bit bus.

Power cables

THE BEST (read, easiest) place to start looking for the cause of a drive problem is the power connection – this is the four-wire cable adjacent to the drive's main data cable. First of all, make sure that the connector is securely seated in its receptacle. Next, try wriggling the connector around slightly, to see if there is a loose contact.

If the drive starts working, then you've found the problem. Don't leave it like this though, because the connector could just as easily come loose again, and if it becomes intermittent, the continual starting and stopping of the drive will put a greater-than-normal stress on the components in it. Many computers have more drive power cables than drives, so it could be just a case of plugging an unused cable into the drive. If this works, tie a knot in the bad cable, so that you remember that there is something wrong with it when you open up the case again in six month's time.

Of course, if you ever need this cable in the future (when you add another drive to the system), then you'll have to do something about it then, if you don't rectify the problem now. The obvious solution is to buy a new connector, and solder it in place of the old one on the end of the cable. If you don't like the idea of wielding

Robert O'Rourke is a long-time computer enthusiast who is disappointed in the lack of support the industry offers new comers to PCs, and disillusioned with much of the advice offered by experts. He would be very interested in hearing of your experiences with simple repairs – forward them to his attention at the Office Services address on the Contents page.

a red-hot poker inside your sensitive computer, then a less labour-intensive solution is to buy a disk drive power Y-cable. Just unplug one of the known-good power cables, and insert the Y-adaptor between it and the drive which it was powering, and you instantly have another power cable.

A Y-cable can also be used for adding another drive to the system, if there are no more power cables spare. If you do this, bear in mind that the reason that there aren't any more power cables available, could be that the supply doesn't have the capacity for any more drives – this is particularly true of some small footprint boxes, which use a lower-rated power supply in conjunction to the lower amount of available space for expansion.

If only somebody could come up with a simple way of storing and transferring data that was as cheap as floppies.

If the new power cable misbehaves in the same way as the first, check the new cable on a known good drive – the problem could be with the drive's power connector, not the one on the cable. Should this be the case, you can probably get a replacement from an electronic parts outlet, such as Rod Irving Electronics. However, you should be competent with a soldering iron before trying this.

If you have two of the same type of drive – either hard or floppy – and one of them is faulty, then it is a relatively simple matter to determine whether the problem lies with the drive itself, or the controller or cable. To do this, simply swap the data cables going to the two drives. That is, unplug the cable from the back of the A: drive, and connect it to the B: drive, and vice versa.

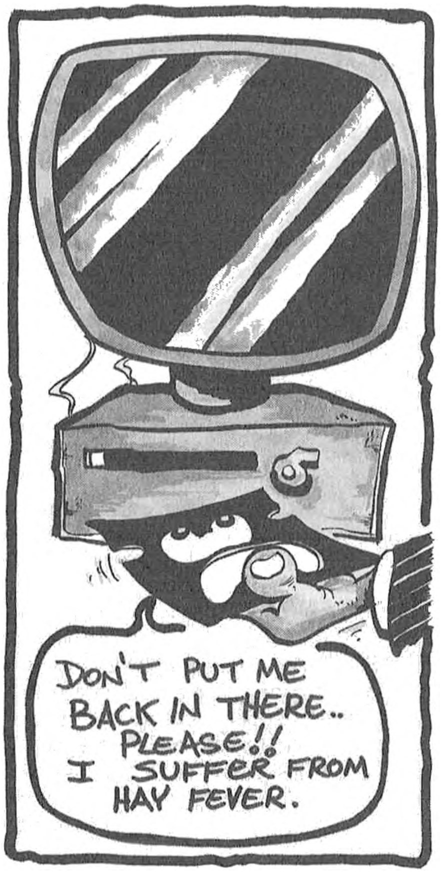
The same applies to hard disks, but if there are separate 20-way data cables (as is the case for MFM, RLL and ESDI units), then these must be swapped simultaneously. Furthermore, if the drives are of different physical types (densities, numbers of tracks, and so on), the CMOS setup will need to be changed to reflect the new positions of the devices, or the DIP switch changed in XT machines.

If the fault follows the physical drive, then the problem is definitely in the drive. However, if the same logical drive causes trouble, then the problem lies either with the controller or the cable. The only real way to narrow it down further is to substitute a known good cable or controller (in that order), possibly from another machine. Bear in mind that the 34-way cables on hard drives are not interchangeable with those used for floppies if either or both have a twist in them between the two drive connectors.

Floppy drives

EVEN MORE problematic for most people than hard disks, are floppy drives. Most of us would gladly be rid of them, if only somebody could come up with a simple way of storing and transferring data that was as cheap as floppies. The reason for the high rate of problems with floppy drives is the same as for hard disks – they are mechanical systems with many moving parts, and thus being prone to failure.

The life of a floppy drive is complicated by three more factors, which between them account for the high failure rate of floppy drives. The first of these is that the floppy disk and drive lack the protection from airborne particles offered to hard disks by nature of their hermetic sealing.


DON'T PUT ME BACK IN THERE.. PLEASE!! I SUFFER FROM HAY FEVER.

Once airborne particles get between the drive heads and disk surface, both suffer more wear than they normally would, and the heads tend to accumulate this debris, forcing them away from the disk's surface, so they don't pick up the signals as well as they should.

Because the heads of floppy drives actually touch the surface of the disk whenever the latter is spinning, there will always be physical wear on both. So don't expect floppy disks to last forever – they will retain their data for a long time if they are stored in a suitably clean environment, but if you use the same set of disks for daily backups for a long period of time, they will eventually fail.

The floppy disk and drive lack the protection from airborne particles offered to hard disks by nature of their hermetic sealing.

Another factor which doesn't arise with hard disks, that can cause problems with floppies, is the removable nature of the latter's media, and the fact that the same disk is not always used in the same drive. The heads will align with the tracks on the disk slightly differently in each drive, and a full-strength signal written by one drive may not be readable on another drive, due to a mutual incompatibility in the alignment of the drives.

The only real cure for this is to get the heads in the problem drives aligned, using a special alignment disk. If this is out of the question, then the only thing to do is to keep the heads on the drives as clean as possible, so that signal losses from that cause are minimised. Also, bear in mind that if you get the heads in a drive realigned, then it may then become incompatible with disks exchanged with yet another machine and, if it was badly out of alignment, it may even be unable to read disks which the same drive wrote to it before it was aligned.

Hard disks

FLOPPY DRIVES are not alone in experiencing problems with head alignment, although the cause is different in the two cases. In floppy drives, as already discussed, the problem is caused by the different alignment in different disk drives.

Of course, the platters in a hard disk are never moved from drive to drive, except as a last resort when a drive has failed and inadequate backups were kept.

The much finer tolerances in hard drives, due to the closer mutual spacing of the tracks, means that the positioning of the heads relative to the tracks on the disk is extremely critical. However, heating of the drive components as the computer warms up, means that the heads and tracks move relative to one another, causing tracks to be written at slightly different distances from the outside of the disk, depending on temperature.

This means that a track written the instant the computer is turned on, will be in a slightly different position when the drive has warmed up. Also, different sectors of the same track could be at different radii, depending on the temperature of the drive when each particular sector was written. As the tracks get further away from the position where the heads are when they try to read the track, the signals retrieved from the heads diminish, and sometimes they cannot be resolved at all.

Rectification of this problem can be solved in a number of ways. The most obvious is to back up the entire drive, re-format the hard disk, and then restore the data from the backup.

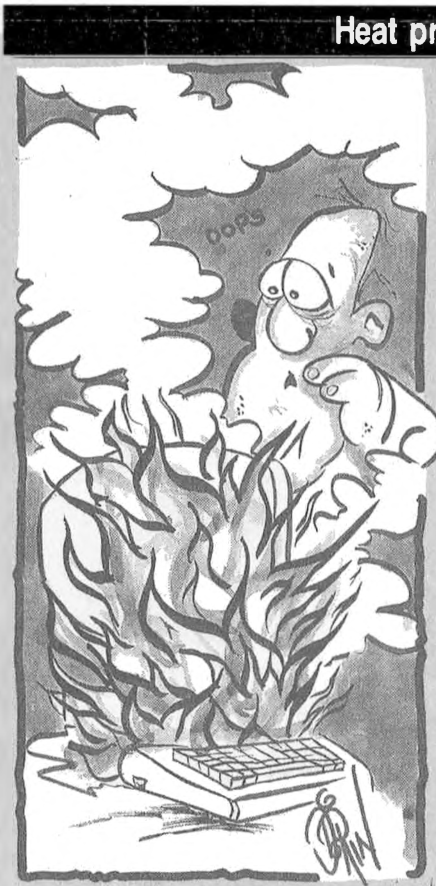
A less labour-intensive method is to use a specialised program which does essentially the same thing, but on a track-by-track basis. It reads the entire track into memory, re-formats the track and then writes the data back to the disk. This saves the time of backing up and restoring the drive, but there is a risk that you will lose data if the power fails in the time between re-formatting the track, and re-writing the data back to it. So, for safety, you should perform the backup beforehand, and if nothing goes wrong (most likely), you have saved yourself from doing the restore. And, you have a backup should something really go wrong later on.

Spinrite is probably the best-known program for doing this, although other utility packages have included similar products in their suites too.

The only drives which need this sort of maintenance are those which use stepper-motors to position the heads, since they use an open-loop head positioning system. There is no feedback from the heads to the positioning circuitry to determine whether the heads have been positioned correctly or not. Voice-coil drives, on the other hand, use a closed-loop positioning system, so as the tracks move as the drive heats up, the head positioning circuitry notices this and compensates for it. Most modern drives (including virtually all IDE, ESDI and SCSI drives) use voice-coil positioning, so you only really have to worry about older MFM and RLL drives, and only those which use stepper motors.

In the final part of this series, we will look at other common computer problems, as well as more advanced testing and repair techniques which can help diagnose computer illnesses. □

Heat problems



ONE PROBLEM which often crops up is an apparently good hard drive, which fails after being used for some time. After turning the computer off for half an hour or so, the drive comes good again. The problem, as you might guess, is heat related, and stepper-motor drives are particularly prone to this.

What is happening is that the components in the drive expand and contract unevenly as the drive heats up and cools down, causing the relative alignment of the heads and tracks on the disk platters to change. If the heads move too far from the tracks, it may not be able to accurately resolve the signals on the disk surface, and then returns data errors.

As a first step, a re-format of the hard disk is the best place to start. Either back up the entire drive, re-format it, and then restore the information afterwards, or use one of the 'on-the-fly' reformatters, such as Spinrite. A low-level format is the best way to go, but don't try this on an IDE drive. Even with a program like Spinrite, it is a good idea to back up the drive beforehand, just in case. If nothing goes wrong, then you save having to do a restore, and score a backup.

If the drive still acts up after doing this, then you'll need to provide some additional cooling for the drive. Before jumping off the deep end, just check the cooling path to make sure that it is not blocked by dust or dead cockroaches.

Some component outlets in the US sell fans especially for the purpose of cooling drives which are getting a bit hot under the collar – I haven't seen them in Australia, but if you ask around, you might find one. Failing this, you can buy small 12-volt DC fans which can be mounted to blow air across the drive. Power is typically derived from a spare drive power connector on the power supply.

If you are installing your own fan, connect the black wire from the fan to one of the black wires in the power cable, and the fan's red wire to the *yellow* one from the power supply – the red wire carries 5 volts to the drive, not 12.

PART 3

FIX IT YOURSELF!

DISK DRIVES ARE not the only components which cause problems in the PC (despite what you may have thought after Part 2 in this series), although they do account for a large proportion. Coverage of all the possibilities would (and has) filled entire books, but here I'll cover the most common and easily-fixed troubles.

Another frequent cause of trouble in AT machines is a flat or marginal battery, which is used to back up the data in the CMOS configuration RAM. The system frequently forgets the time and date, or can't find the hard disk drive: The CMOS RAM stores critical system configuration information, and if it loses power, the BIOS won't know all it needs to about the system.

Most AT motherboards have a lithium battery connected to it via a short cable – the battery itself is often attached with a strip of velcro to the side of the power supply. In these cases, simply remove the battery and hunt around the computer parts suppliers for a suitable replacement. You may not be able to find one which looks exactly the same, but provided it is the same type of battery and, most importantly, the same voltage as the one it is replacing, it should work fine. If you have access to a volt meter, check to see if the voltage is below the value marked on it – if so, it is a sure sign that it is in need of replacement.

Unfortunately, some computer manufacturers, in their seemingly never-ending search of ways to shave a bit more off production costs, have seen fit to solder the

In the final part of his PC maintenance series, Robert O'Rourke discusses memory problems, expansion card pitfalls and diagnostics.

battery directly to the motherboard. Even finding it can necessitate removing the motherboard from the chassis (which in turn involves removing half a dozen other components first), and you then have to wield a soldering iron near the motherboard. Perhaps that is an indication of their confidence in the life-expectancy of their products?

Memory problems

THERE ARE two types of electronic memory in a computer – read-only memory (ROM), and random-access read-write memory, or RAM. The former is usually present on the motherboard, and on some expansion cards. The only time it causes problems is when two expansion cards are plugged into the computer, both of them having ROM chips which try to occupy the same address. This is pretty rare, and also easy to diagnose – the computer worked before the last board was plugged in, and didn't work after.

On the other hand, RAM can be the cause of a lot of problems in PCs, largely because there are a variety of different ways that it can be addressed by the processor, and even more differences in the way additional memory is added to different computers. We won't go into the differences between the different types of memory in detail – to do so would take up the entire article, and Stewart Fist wrote a good explanation of the different types in his article in the May 1990 issue of *Your Computer*.

The type of RAM used in PCs is known as *dynamic RAM* (DRAM), because the information stored in the RAM chips needs to be replenished regularly, or it will 'forget'. Static RAM (SRAM) does not need this *refresh* cycle, but because of its far greater complexity, chip makers cannot fit nearly as much storage into a given space as in dynamic RAM, which is why it isn't used except for very high speed applications such as memory caching.

One of the most common causes for memory problems in PCs is caused by using RAM chips which are too slow for the speed of the processor. When the processor requests the contents of a particular location in RAM, it doesn't wait long enough for the RAM chip to present the data, and so the data read by the processor is incorrect.

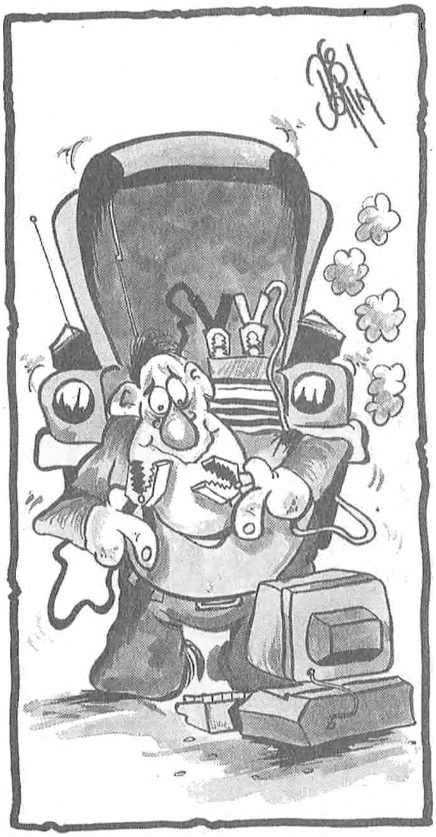
When designing the PC, IBM built in a protection system to guard against memory errors, by building a parity check into the memory system. Dynamic RAM chips are typically 1-bit wide, although 4-bit chips have also become relatively common of late. This means that in order to store data which is byte-wide (as was the case with the original PC and XT machines), eight chips were required. However, IBM added a ninth chip to each bank of memory, which stored a parity check for the other eight bits.

The parity bit is calculated and written at the same time as the 8 data bits, and when a given location is read from memory, the parity is re-calculated from the read data, and compared with the parity value stored in that chip. If the two values do not match, then an error has occurred in either the read or write operation, and usually the system is halted at this point.

Parity errors are usually the result of too-slow RAM chips, but determining which chip is at fault can be difficult. The simplest solution is to slow down the processor, either by reducing the frequency of its clock signal, or inserting wait states, where the processor waits for one or more extra cycles before reading the data from the RAM chips, to give it time to stabilise. Of course this is only a bandaid solution, as the performance of the entire system will suffer because of one RAM chip's sluggishness.

One way to test RAM chips is to swap them one by one with a known good one until the problem goes away. However, if the chip's performance is really borderline, then the fault will be intermittent in nature, and you may never be sure that you've licked it. Plugging and unplugging chips repeatedly is a guaranteed way to break their legs off. Also, if the computer uses SIMMs (single-inline memory modules), then you have to replace the entire module, which costs a lot more than a single DIP. However, it is really the only way to do it, unless you have the equipment to test the speed of RAM chips.

I had originally planned to limit my discussion of testing RAM chips to a simple trial and error process – the equipment required to test the speed of RAM chips was simply not readily available to the average

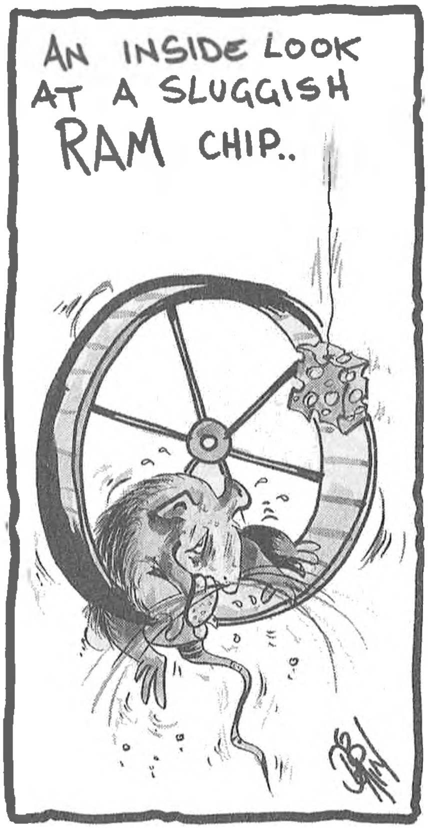


user. However, Microgram has just started importing a little gadget which does just that – tests RAM chips. The chip under test is plugged into the box, and is tested at progressively faster speeds, until it fails. The highest speed at which the chip passes the test is of course the speed of the chip. The tester can test DIP, SIPP and SIMM type RAMs, and Microgram – (043) 34 1544, fax (043) 34 1334 – supplies all necessary adapters.

If you have parity error problems, and have access to such a tester, then it is simply a matter of plugging each of the RAM chips (or SIMMs) into the tester one by one, and setting the testing speed to the speed marked on the chips. If any of the chips fail the test, then they are suspect. If all of the chips pass, it is a good idea to re-run the tests with a slightly faster speed, just in case the bad chip is a bit borderline.

RAM chips usually have their speed marked on them, as a suffix to thè type number. For example, a chip marked 41256-10 is a 256K bit chip with a rated speed of 100 nanoseconds – the suffix is actually a tenth of the actual access time.

Once you have isolated the problem to a particular RAM chip, then all you need to do is to get a suitable replacement chip, and plug it in. When doing so, make sure that the chip number and speed are the same as the one which the new one is to replace. So, a 100ns 4464 should only be replaced with a 100ns 4464, although a faster part should pose no problem. Just make sure it isn't slower.



Determining what speed memory chips should have in a given system is not easy, as it depends on a variety of factors, including the type of processor used, the number of wait states (deliberate delays in memory access cycles to cope with slow memory), and whether other tricks are used such as caching or interleaving. Really, all I can suggest here is to follow the specification of the manufacturer of the motherboard, or use chips that are at least as fast as those already in the machine.

Expansion cards

MANY PROBLEMS ARISE not during the day-to-day operation of the machine, but when a new addition is made to the system. This is usually the result of an incompatibility between the new item, and one which already exists in the system. For example, a second floppy disk controller will certainly conflict with the one already in the system, unless one or the other has the ability to reside at a second (non-conflicting) address.

Such a conflict will manifest itself at the very least with both of the controllers not working at all and, most probably, with an error message being generated during the power-on self test sequence. However, there are other potential sources of conflicts, whose causes are not as apparent, and whose effects do not manifest themselves immediately.

Multi-function cards are often a cause of trouble, since each of the functions of the card has the potential to conflict with other devices in the system. It is not unusual, for example, to see a single half-length card with an IDE hard disk interface, a floppy drive controller, a parallel port, two serial ports, and a games port. If any of these items already exists in the system – either on the motherboard or on another expansion card – then there is the possibility of a conflict.

Often a multi-function card will have one of two ways to avert conflicts. One is to have a switch or jumper to disable each part of the card, and the other is to change the address of the offending device. The former is the least troublesome, but it means wasting part of the circuitry on the card. However, in some cases, it is the only option. For example, if you have two cards with games ports on them, then one of the ports needs to be disabled, since the PC can only address one games port.

However, the PC can support up to three parallel ports, and four serial ports, so if serial or parallel ports are the cause of the problem, you can often change the address of either the new or existing ports so that they do not conflict.

The type of conflict discussed above is an address conflict, where two devices try to share the same space in the I/O address space of the microprocessor. There is another common cause of conflict in computer systems which can be much more mysterious – an interrupt conflict.

Interrupt lines are used by various peripheral devices, such as disk controllers and serial ports, to signal the processor that they need to be serviced in some way. There are nine interrupt lines in the XT, and sixteen in the AT, each of which is allocated a different function.

When a signal arrives on a particular interrupt line, the processor completes its current operation, and then jumps to an address in memory allocated to that interrupt line. That address, called an *interrupt vector*, is set up by the software which is associated with the hardware attached to the interrupt line, so that it will be notified when the hardware requires input or output. However, if two devices share the interrupt line, the interrupt vector will be set twice, so the first piece of software loaded

will never see any interrupts – they will all pass to the second piece of software.

The trouble is, these interrupts could originate from either of the two devices attached to the interrupt line, one of which is designed to work with that software, and one that isn't. This sort of thing usually results in the software crashing, when it receives an interrupt which it doesn't know how to handle. It is possible to have an interrupt line shared amongst several devices, but both the software and hardware must be designed with this in mind.

Interrupt conflicts usually only arise if you are adding an 'unusual' piece of hardware to a computer system, such as a scanner interface, or a third and fourth serial port. In most cases, resolving the conflict is simply a case of selecting an unused interrupt line on the new card, and setting the software appropriately.

Another hazard to watch for is the so-called eighth slot in the XT. This slot has slightly different timings to the other slots, so a card which works perfectly in other slots may not work in this slot. If there is another free slot, try using that for the new card, or plug another card into slot eight, so you can put the new ward in another slot. Some cards, such as graphics cards, have a special slot eight jumper, which allows the card to be used in this slot. Make sure the jumper is set correctly if using this slot.

POST tests

NO, POST TESTS don't get their name because they happen after something – in fact, they are run before the computer does anything else. POST, in fact, stands for 'power-on self test', and is a small program contained in the BIOS chips, which tests the integrity of the computer hardware before it attempts to run any more complex software.

This self-tests the various components of the system, and can perform tests which would be simply impossible using a diagnostic program loaded from a disk. How can you run a diagnostic program if the disk controller is stuffed (you can't even load it) or the memory system is non-functional? There is no place to store the program. Because the POST routines are contained in the BIOS ROM chips, they are always present, so they can be run on a system with major faults, and help diagnose these faults.

The trouble with the POST, of course, is that it *does* require some functionality to be present in the computer for the routines to run at all – the processor and all the vital components which support it need to be present, and in working order, as does the BIOS itself. However, even with these limitations, a large part of the system is able to be tested for potential reliability before the user trusts his valuable spreadsheet data to its judgment. Chances are, any problem with the aforementioned vital components, will result in a total absence of anything resembling life from the computer, leaving one in no doubt that something is amiss.

You are probably already familiar with some of the POST messages – if you have ever turned a computer on without the keyboard plugged in, you'll know what I mean. The POST performs a relatively brief test of the vital system components, and reports any problems it finds. So, provided the display system is in working order, the POST messages should go some way toward helping you isolate the problem.

If an error is reported in some component on the motherboard, like the DMA controller, then it's probably time you sought expert help.

But, what happens if the problem is with the display itself? How does the computer tell you that it's sick, and more importantly, what the problem is? In these cases, the POST uses the only output device it knows it can count on – the speaker. One long beep, followed by two short ones (the letter 'D' in Morse code) indicates a display failure. Of course, if those beeps aren't emitted, and the display is still blank, the problem could still lie in the video card, but the monitor is a more likely culprit. Again, a process of swapping components with a known good computer is the only real way of being certain.

Diagnostic programs

IN ADDITION to the POST routines built into the ROM chips, there is a variety of diagnostic programs on the market which can aid you in isolating the cause of problems with your computer. One of the most comprehensive is Checkit, from Touch Stone Software, distributed in Australia by Technisoft, (075) 91 2499, fax (075) 91 1639.

Checkit includes a wide range of tests for various system components, including hard and floppy disk drives, I/O ports, memory (base and extended), and motherboard components, as well as peripherals such as a keyboard, mouse and printer. In addition, it can display maps of the memory used by various programs, the owners of the various interrupt lines and DMA channels, and even the contents of the AT's CMOS RAM table.

Of course, utilities such as Checkit are only useful if the computer is working well enough to actually load and run the program – if the processor is completely stuffed, no diagnostic program will be able to help you.

The end

IN THIS series, we have barely scratched the surface of computer maintenance. Software problems, for example, haven't been discussed at all, and the hardware diagnoses have been limited to the board level. Delving deeper than that into the PC's insides requires a fair bit of background knowledge, and is definitely not a job for the inexperienced.

As with any complex system, once you get to know the functions of the various component parts, and take a logical approach to searching for the problem, there is little you can't fix. □

Robert O'Rourke is a computer enthusiast with a well-developed disappointment in the lack of service the industry offers new-comers to PCs, and he has become disillusioned with much of the advice offered by experts. He would be very interested in hearing of your experiences with simple repairs – forward them to his attention at the Office Services address on the Contents page.

An Introduction to Pascal - Part 1

Pascal is probably the best language to learn programming with – almost by definition, a program must be well-organised, structured, modular and easy to maintain and follow. In the first of a three-part series, John Taubenschlag outlines the basics . . .

PASCAL, ONE OF the more recently developed programming languages, was created with three main objectives in mind – first, for it to be used as a powerful problem solving language; second, for it to be easy to learn and utilize; and third, to facilitate modular programming techniques.

Pascal embodies some of the latest concepts of programming techniques, including the ability to define one's own data structures. Because of the way the language is defined, programming code must be very organized which makes it easy to follow.

Listing 1 presents a simple Pascal program which reads in a positive number from the keyboard and displays the number which it reads, its square and cube. To get a feel for the language, let's work through the program – the first word in any Pascal program is 'program' which tells the compiler that the program starts at that location. The next characters which appear are the name given to the program – in this case, 'numbers'. The only two rules to be followed when naming a Pascal program are that *1)* the name may not have any blanks in it; and *2)* that the name is not a reserved word.

A 'reserved word' is the name of a command or instruction in Pascal (these include: write, writeln, readln, and, or, not, true, false, begin, end, boolean, char, text, real, integer, input, output, procedure, for, else, if, case, then, while, repeat and until).

Because Pascal is a very ordered language, long before typing in a program, you must have planned all the functions which the program will carry out. The next statement after the name of the program (within parentheses) gives the names of the files the program will use. Also, whether it will input, output or both. Thus, (input,output) tells the compiler that the program 'numbers' will input and output data but will not use any external files. There are many devices to which data can be sent; these include the screen (VDU), printer, disk drives or a tape drive. In this case, the data will go to the screen, since none of the other peripherals have been opened for access. To do this it is necessary to use special commands depending on the computer you have.

After every statement in the program, with the exceptions of loop, case, begin and a few other statements, each line must end with a semicolon. To make a comment or remark in a program, you must enclose it with parentheses and asterisks –

```
(* This is a comment!*)
```

Unlike Basic, when programming in Pascal, you must know before hand all the variables which are to be used, and also what they will be used for. The Var statement, tells the Pascal compiler that all the variables are about to be listed – this must be done after the program name. (A compiler is a program that accepts input from a high level language like Pascal and 'compiles' a program in the machine language that the computer understands.) In Listing 1, num1, num1sqr and num1cube are all the variables to be used in the program. The word 'real' means that all those variables will take real number values (as opposed to integer values). To separate a variable from its identifier, a colon is used.

If more than one variable is in the same group, such as a group of variables which will all take real values, they must be separated with commas. A variable name may not be a reserved word. Other variable identifiers include integer, char and boolean (an integer variable may store a whole number, while a char variable stores a single character and boolean is a flag variable which can take the values of true OR false).

```
program numbers (input,output);

(* Author    : John Taubenschlag     *)
(* Date      : 27th July, 1987      *)
(* Purpose   : This program will ask for *)
(*             a positive value from   *)
(*             the keyboard, and it will *)
(*             find its square and cube. *)
var
num1,
num1sqr,
num1cube : real;
(* Procedure I - Reading the information from a user*)
procedure obtaining;
begin
 writeln (output,'       Number Finder');
 writeln (output,'       -------------');
 writeln (output);
 writeln (output);
    square');
 writeln (output,' This program will find the :
 writeln (output,'             and cube');
 writeln (output,' of any positive number that you
    type in.');
 writeln (output);
 writeln (output);
 writeln (output,' To type it in, use the ten
    number keys at the');
 writeln (output,' top of the keyboard. After the
    number, press);
 writeln (output,' the return key.);
 writeln (output,' EG :  >29 <RET>');
 writeln (output);
 writeln (output);
 writeln (output);
 writeln (output,' Now, please type in your number.');
 writeln (output);
 write  (output,' >');
 readln (input,num1);
 writeln (output);
 writeln (output);
 writeln (output,' Your number was accepted.');
 writeln (output);
 writeln (output);
 writeln (output);
 writeln (output);
end;
(* Procedure II - Manipulating the data *)
procedure manipulating;
begin
 (* calculating square of number *)
   num1sqr := num1 * num1;
 (* calculating cube of number *)
   num1cube := num1 * num1 * num1;
end;
(* Procedure III - Printing output to the user *)
procedure printoutput;
begin
 writeln (output,' Results :');
 writeln (output,' ---------');
 writeln (output);
 writeln (output);
 writeln (output,'******************************
    ******');
 writeln (output,' Number *  Square *  Cube  ');
 writeln (output,'******************************
    ******');
 writeln (output,'     *       *      ');
 writeln (output,num1:10:2,'*',num1sqr:11:2,'*',
   num1cube:11:2);
 writeln (output,'******************************
    ******');
 writeln (output);
 writeln (output);
 writeln (output,'    End of Program.');
end;

(* Main Program *)

begin
 obtaining;
   manipulating;
   printoutput;
end.
```

***Listing 1.** A simple Pascal program which reads in a positive number from the keyboard and displays the number which it reads, its square and cube.*

The textbook method of writing a Pascal program is to write it in sections – called 'procedures' in Pascal. To begin a procedure, simply type 'procedure' followed by a name. In Listing 1, the name for the first procedure is 'obtaining'. Once again, the procedure name may not be in the reserved word list. The name of the procedure is followed by the word 'begin' which tells the compiler that the procedure begins here.

As can be seen in the program structure, indenting is used extensively to make the program easier to read. Convention indents for each level of the program, so there's an indent after 'begin'; if there were a loop in the program, that would be indented again, and so on.

Writeln is the equivalent of Basic's PRINT statement – it's the command which instructs the computer to output data. Its format is as shown in the listing – note the order of the punctuation. Writeln will print the text followed by a carriage return. If the carriage return is not wanted, 'write' is used with the same format. Thus, the write statement will print the text, but will remain on the same line waiting for the next command.

The readln statement is used to input data. In the listing, the readln statement will input data from the keyboard and store it in num1. Since num1 is a real variable, the program will expect a number; if a non-number character is typed, the program will crash (stop with an error). Each procedure must end with 'end' to tell the compiler that the procedure has finished.

Procedure II

IN LISTING 1, Procedure II shows how variables can be assigned a value. First comes the variable which is to have a value assigned, then := which is the way things are equated in Pascal, and then the value or another variable which has already been assigned a value. Here variable a is assigned the value of the variable b plus the value 42 –

```
a := b + 42;
```

Other arithmetic and logical (such as AND) functions are carried out similarly. Pascal's other functions include –

ABS(x) to find the absolute value of *x*;

EXP(x) to raise the natural logarithm *e* to the *x* power;

ODD(x) returns 'true' if *x* is an odd number;

ROUND(x) transforms a real number to an integer by rounding it off to the nearest whole number;

SQR(x) finds the square of *x*;

SQRT(x) finds the square root of *x*;

TRUNC(x) transforms a real number to an integer by chopping off everything after the decimal point; and

<, > and

<> are the familiar 'less than', 'greater than' and 'not equal to' functions.

Procedure III

IN PROCEDURE III, 'printoutput', the results are printed out. This is the format which the variable output statement has –

```
writeln (output,num1:10:2,'*',num1sqr:11:2,
   '*',num1cube:11:2);
```

Note the variable part of this statement: 'num1:10:2' means that the variable num1 will be printed within 10 spaces, starting from the right of the screen. Of those 10 spaces, 1 is for the decimal point (since num1 is a real variable) and another 2 are for numbers to the right of the decimal point. Within the other 7 spaces, the integer part of the number is printed. This function is used to justify all the numbers so that the output is more legible.

Pascal Past

PASCAL WAS developed primarily as a teaching tool for programming – the need was for a high-level ('English-like') language that would be efficient, complete in itself, well-ordered ('structured') and easy to learn.

Other languages are used to teach programming, particularly Basic which is easy to learn and will run on small computers; Basic's principle disadvantages are that the rules of usage (the 'syntax' of the language) are severely limiting and the early versions of the language didn't lend themselves to modular, complex, well-ordered programs. (When developing a program to solve a complex problem, it's essential that the problem be broken into smaller ones which can then be solved individually; these individual solutions – modules – are then combined into a single, complex program.)

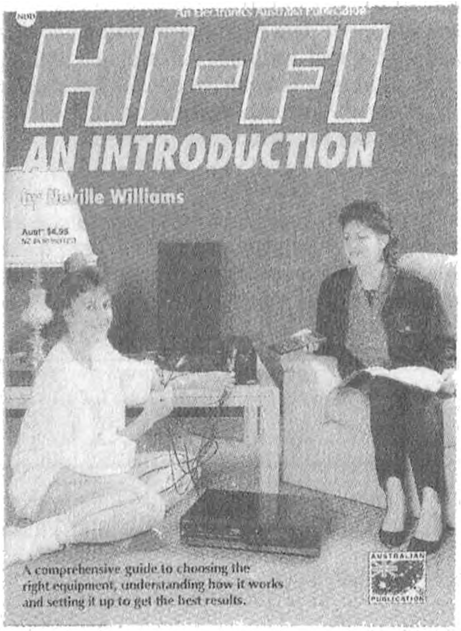
Pascal (named for the 17th century French mathematician Blaise Pascal) was 'invented' by Niklaus Wirth in the early '70s (Wirth later invented Modula-2 as a more powerful alternative to Pascal).

Similar to any other high-level language in wide-spread use, many versions have been developed from Wirth's Standard Pascal – these all implement the Standard version, but incorporate features suited to a particular computer or operating system. The first popular version of Pascal for microcomputers was UCSD (University of California at San Diego) Pascal – it's probably the most widely used version for teaching today.

Its popularity as a teaching tool is waning, though, because an even more efficient version (with powerful editing features) was developed specifically for micros by Frank Borland – Turbo Pascal. For commercial use, it's now much more-widely used than any other version and it's finding its way into high schools and universities as a teaching tool.

If one compiles and runs the program as it is, it will do nothing because, while all the procedures have been defined, none have been called – that's the purpose of the main program. When the

Continued on page 68.


An Electronics Australia Publication
HI-FI
AN INTRODUCTION
by Neville Williams
Aust* $4.95
A comprehensive guide to choosing the right equipment, understanding how it works and setting it up to get the best results.

Introduction to Pascal - Part 2

In our April issue, John Taubenschlag presented the basics of Pascal with a simple program and explained in detail how it worked. Now, he develops a new program demonstrating the use of more complex commands and processes.

THE PROGRAM discussed in this part of our tutorial is designed to alphabetically sort a list of characters, which have been typed in from the keyboard. The process used for rearranging the characters will be a bubble sort. To be able to understand this program, we must first take a look at the new statements used – const, type, ary, arrays and the if-then statement.

At the beginning of any Pascal program, all the variables must be listed in the var section, and be assigned a data-type (integer, real, char, whatever). However, Pascal allows you to do two more things. First, you can create a constant. In our example, numofletters has been set to the value of 10 (note: you cannot change the value of a constant during a program). These are examples of different constants which may be created –

```
const
   numofletters = 10;
   word = 'hello';
   phrase = 'I do not understand.';
   blank = '            ';
   pi = 3.14159;
   low = -1000;
   high = 2500;
```

Second, when programming in Pascal, you can create your own data-types, such as an array. In the program, ary is not a variable, but a data-type, therefore, Pascal will recognise ary like all the other data-types (integer, char, real and so on); ary is defined as an array which can hold a definite number of single characters. In this particular case, the number it can hold will depend on the value of numofletters, because the array is defined between 1 and numofletters (which is set to 10).

An array is like a normal variable with the exception that it has a number, within square brackets , attached to the variable name. For example: letter[3], letter[5] and letter[8] are all variables from the same array, which may all have different values. Although in Listing 1, the upper-limit of the array is a constant, this is not always the case. Therefore, having defined ary in the type section, you can now assign a variable with the ary data-type. This step is done in the var section, as shown below –

```
var
   letter : ary;
```

These are further examples of arrays –

```
type
   horse = array[1..5] of char;
   questions = array[5..15] of boolean;
   numbers = array[1..1000] of integer;
   numbers2 = array[1..250] of real;
```

Lastly, there is a strict order which must be followed, when using the const, type and var commands – const comes first followed by the var section. This means that if you define a constant in the const section, you can use it in both the type and var sections. However, if you define a data-type in the type section, you can use it in the var section but not the const section. Lastly, if you define a variable in the var section, you can not refer to it in any of the other two previous sections.

The if-then statement is a very commonly used command in most programming languages, and in Pascal it is no exception. Its use is to set-up a condition in the program. To demonstrate it, we can use an everyday example: IF I have more than fifty cents, THEN I can afford to buy an apple. In Pascal that phrase would be written as –

```
if money ) .5
then
   afford_apple := true;
```

The if-then statement can be further expanded to the if-then-else statement –

```
if money ) .5
then
   afford_apple := true
   else
     afford_apple := false;
```

Thus, if money is greater than half (fifty cents), afford – apple is set to true, otherwise it is set to false (note there is no semicolon before the else statement.

When setting-up conditional statements in Pascal, there are three key words to work with : *not*, *and* and *or* – 'not' makes the variable used in the conditional statement have its opposite value. For example –

```
flag := true;
while not flag do
begin
  writeln (output, 'Hello');
  flag := false;
end;
```

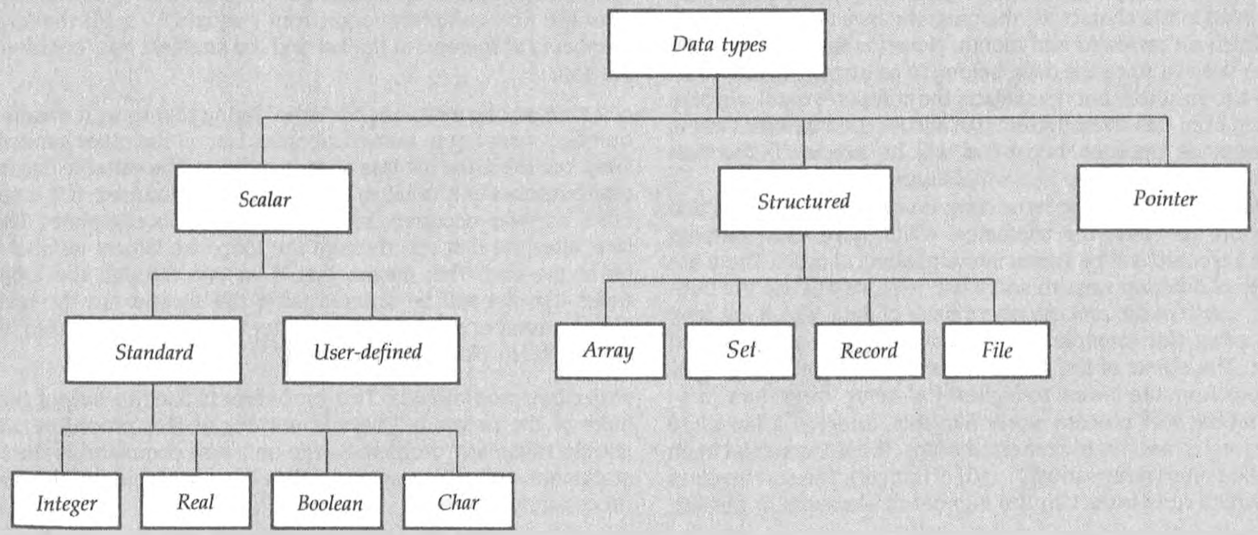
In this example, the loop will be repeated until the variable flag is set to false. This is because 'while not flag', instructs the com-

```
program bubble_sort (input,output);

(* Author   : John Taubenschlag     *)
(* Date    : 25th August, 1987      *)
(* Version  : 1.2               *)
(* Purpose  : This program will ask for *)
(*       ten characters, and then *)
(*       using a bubble sort, will *)
(*       arrange them in alphabet- *)
(*       cal order.          *)
const
   numofletters = 10;

type
   ary = array[1..numofletters] of char;

var
   i,
   count,
   done   : integer;
   flag   : boolean;
   store  : char;
   letter  : ary;

(* Procedure I - Initialising the variables *)

procedure initialise;
begin
   done := 0;
   count := 1;
   flag := true;
end;

(* Procedure II - Reading the characters *)

procedure readin;

begin
   writeln (output,'Type in ten letters, each followed
   by the return');
   writeln(output,'key.I will then sort them for you.');

   for i:=1 to numofletters do
   begin
     write (output,'Type in letter number ',i,' : ');
     readln (input,letter[i]);
     writeln (output);
    end;
end;

(* Procedure III - Sorting the data *)

procedure sort;
```

```
begin
  while flag do
  begin
    flag := false;
    while count ( (numofletters - done) do
    begin
       if letter[count] ) letter[count + 1]
       then
         begin
            store := letter[count];
            letter[count] := letter[count + 1];
            letter[count + 1] := store;
            flag := true;
         end;
       count := count + 1;
    end;
    done := done + 1;
    count := 1;
  end;
end;

(* Procedure IV - Printing output to the user *)

procedure printoutput;
begin

    writeln (output,' Results :');
    writeln (output,' ---------');
    writeln (output);
    writeln (output);
    writeln (output,'These are the letters in
    alphabetical order :');
    writeln (output);
    writeln (output);
    for i:=1 to numofletters do
      writeln (output, letter[i]);
    writeln (output);
    writeln (output);
    writeln (output,'   End of Program.');
end;

(* Main Program *)

begin

    initialise;
    readin;
    sort;
    printoutput;


end.
```

***Listing 1.** This example program is designed to alphabetically sort a list of characters typed in from the keyboard using a bubble sort – it also introduces a number of new statements: const, type, ary, arrays and the if-then statement.*

Data types



DATA IS ANY information the computer is given to perform operations on or otherwise manipulate – the unit of data is defined as a *data element*. Generally the most basic data elements are numeric (comprising numbers) and characters (text), but in keeping with the tightly defined structure of Pascal, data types are much more stringently defined than that in the language.

The three major classifications of data types are *scalar, structured*, and *pointer*. Scalar data types have their data elements ordered, that is, each one is defined as being equal to, less than or greater than the other elements. It's obvious that numeric data such as *real* numbers and *integers* are scalar, but – to have meaning, characters are just as 'ordered' as numbers, so *character* data elements are also scalar. The *Boolean* data type is the simplest that Pascal works with – the elements can only be TRUE or FALSE; by definition in Pascal, FALSE is always less than TRUE, so Boolean data is also scalar.

Integer, real, Boolean and character are all *standard* data types, that is, their definitions are all predefined in Pascal. For a particular application, however, it may be convenient or necessary to create a *user-defined* data type for a specific purpose – one of the most often employed user-defined data types comprises the months of the year (note that these are ordered, so this is also a scalar data type).

Structured data types are complex in that they are built up from scalar data types – as an example, a *record* might consist of a company's name, address and phone number, all of which are either character or numeric, that is, scalar. A *file* can be thought of as a collection of records, while an *array* is two or more logically (in the Boolean sense) related data elements and a *set* is a user-defined collection of records – all of these are built up from scalar elements.

Pointer data types are much more complex – suffice to say that they specify (point to) a memory address where the data required can be found (or stored for later retrieval).

puter to repeat the loop until flag equals false. The second key word, and, may be used in the following way –

```
if (x > 10) and (x <= 15)
then
   writeln (output,'Your test grade is B.');
```

– which means, for the if statement to be true, there are two conditions which must be met simultaneously, namely that x is greater than 10, but less than or equal to 15. The third key word is or. It can be used as follows –

```
if (x <> 10) or (y = 19)
then
   limit_reached = true;
```

The if statement will be true, if at least one of the conditions is met. Thus, if x does not equal 10, or y equals 19, the variable limit reached will be set to true. The brackets which appear in these statements are the same as those used in mathematical equations so the conditions within the brackets are processed first.

Brackets are also used to simplify the expressions so they are easier to read and understand

Finally, the three key words have a processing order. This is like in mathematics, where multiplication is done before addition. The order is not, and, followed by or.

Procedures

NOW, LET'S go through the program, procedure by procedure, and see,exactly what each section does –

Procedure initialise: This procedure gives the variables in the program their initial values. Unlike Basic (where all variables have pre-set values), when a Pascal program is run, all the variables are undefined. Thus, if you try to use them in a statement where they must have a value, such as an if-then, while or writeln command, the program will crash (stop with an error).

Procedure readin: As the name indicates, this procedure reads in the data from the keyboard (see Procedure II in the example listing). To read in the characters, the program uses the readln statement, which we reviewed last month. Nevertheless, this time the variables used to store the data, belong to an array. The first command is a loop statement; it instructs the computer that i will take the values from 1 to numofletters (10) and for each different value, the commands between begin-end will be executed. The first command within the loop is a write statement.

Procedure sort: This is the most complex section of the program – Procedure III. Here, the characters which have been entered from the keyboard will be sorted into alphabetical order. There are a number of different ways to sort a list; here we will use the bubble sort, a systematic process where pairs of data, which are next to each other (for example letter[3] and letter[4]) are examined together. The object of the sort is to have an ordered list of data which runs from the lowest to highest ('a' being lower than 'b').

The bubble sort process works like this: imagine a list of 10 numbers which are not in numerical order. The list is stored in an array called num (num=array[1...10] of integer). The sort involves a loop which runs from 1 to the number of elements in the list. During each run through the loop, pairs of numbers are examined together. Thus, num[1] will be compared to num[2]. If the first is larger than the latter, they will be swapped, otherwise they will be left as they were. Then, num[2] is compared to num[3], and the same conditional swap is applied. The significance of this, is that after the first complete loop (from i equals 1 to 10) the largest number is at the end of the list and the smallest has 'bubbled' to the top.

If at least one swap has occurred during that loop, it means the numbers were not in numerical order, but on the other hand, if no swap occurred, the list was already ordered (the variable flag is for that purpose – if it is set to true, a swap has occurred; if it is set to false, no swap occurred, so the sort process is complete). Therefore, after the first run through the loop, the largest number will be at the end. This means that if we run through the loop 10 times, the list will be ordered (after the second run the second highest number will be in place, after the third run the third highest will be in place, and so forth).

Procedure printoutput: This procedure is used to output the results of the program. There is nothing in this procedure which should cause any problems – the only new command is the loop statement with the array, which has been explained in the readin procedure as above. □

PASCAL

Continued from page 64.

compiler reads a begin, which does not follow a procedure heading and is not part of a loop, it knows that it has arrived at the main program. To call a procedure, type its name followed by a semicolon; to end the main program, the end statement is used, followed by a full stop instead of a semicolon. The rule is that the procedure must be defined before it is called – that's is why the main program comes at the end.

Loop it!

AN EASY way to make the computer do a repetitive process, other than typing the instruction many times, is with the use of a loop statement. There are three ways of doing this in Pascal. First, there is the FOR statement –

```
for i:=1 to 10 do
begin
   write (output,'hi ');
   writeln (output,'there!');
end;
```

Notice how there is a begin and end around the part which has to be repeated. This section of a program will print 'hi there!' under each other 10 times. It is important to note that the variable i will take the values from 1 to 10 (variable i must have been defined in the var section, at the beginning of the program, as an integer).

Second, there is the WHILE statement –

```
while i<10 do
begin
   writeln (output,i);
   i := i + 1;
end;
```

This will print the number from the initial setting of i to 10. Note that within the begin-end loop, there must be an increment of i such that the loop is not endless and i can reach the number 10.

Third, there is the REPEAT-UNTIL statement –

```
repeat
   readln (input,num1);
until num1=999;
```

Here there is no need for a begin and end. The expression(s) within the REPEAT-UNTIL loop will be performed until the specified test to stop it is met – in this case the number which is read must equal 999. The main difference between the loop statements, is that with the FOR and WHILE loops, the test for continuance is done at the beginning while in the REPEAT loop, the test is done at the end.

In Part 2, we will develop a new program which will demonstrate the use of more complex commands and processes, such as arrays, constants, creating data-types, loops, and the if-then statement. □

Introduction to Pascal - Part III

In May, John Taubenschlag demonstrated the use of complex commands and processes in Pascal. But – how do we read data from the disk drive?

ONE OF THE most important aspects of a computer system, whether it is a microcomputer or a mainframe, is its ability to store and retrieve information. Since the efficient use of storage devices is of the utmost importance when using a computer system, let's look at the basics of reading data from the disk drive.

For an example program, we'll simulate the Logo Turtle, a simple robot, which obeys a limited set of instructions (see Listing 1). The instructions used to communicate with the turtle are –

LEFT which rotates the turtle 90 degrees to its left;

RIGHT which rotates the turtle 90 degrees to its right;

FORWARD *x* which moves the turtle forward *x* units in the direction it is facing; and

STOP to end the movement of the turtle.

Using these four instructions, it will be possible to control the movement of the turtle. However, in this program, instead of typing in the instructions one at a time when prompted by the computer, they will be typed all at once into a file. The advantage is that if there are 100 instructions, the program can be run several times without having to type all 100 instruction over and over again. When the program is run, the file will be read, and after simulating the turtle's actions, its displacement calculated.

This is the format which the file, named TURTLE, must have –

LEFT FORWARD 19 RIGHT RIGHT FORWARD 4.5 LEFT FORWARD 9.45 LEFT RIGHT FORWARD 3 LEFT LEFT STOP

When the computer reaches the STOP instruction, it will compute the displacement of the turtle. There is no limit to the number of instructions which may appear in the file. However, there are four important characteristics to note: first, there is a single space between each command and/or digit; second, the number of units to go forward could be a real number or an integer; third, when creating the file, do not press the return key at the end of each line, rather let the editor wrap the text around to the next line; and fourth, the file must be typed in upper case.

New commands

BEFORE looking at Listing 1 in detail, let's cover the new commands it includes – there are four commands which have not been covered yet: infile, text, reset and the case statement.

Infile is not a Pascal command, rather it is a variable which was-created for the purpose of allowing Pascal to open a file on the-disk. Thus, infile is the name by which the file on disk will be referred to within the program. It must appear in three places –

First, it must be placed in the first line, within round brackets, after the program name. This tells the compiler that a file will be opened on disk. It does not necessarily have to be called infile, it could be called teacup, turtlefile, whatever.

Second, it must appear within the var section. Here, the file variable (infile), must be given a data type, in other words, a description of what data the file contains (see Part II for a discussion on data types).

Lastly, it must be found within the program. Here is where the file on disk is actually opened. Since it is necessary that the file be opened before the program tries to access it, it's a good idea to place the reset command in the initialise procedure at the beginning of the program with –

```
reset ( a , 'b' );
```

Where 'a' is the variable file name in the program (in this case infile), and 'b' is the file name on the disk (in this case TURTLE).

After the file is opened, the program can read from it. To do this, the read and readln statements are used. If a single character is to be read, we use –

```
read (infile,ch);
```

Where 'ch' is a variable defined to hold a single character. When data is read from disk with the read statement, an imaginary file pointer moves to the character immediately after the one read. Therefore, the next letter you read, will be the one which the pointer is on. However, if you use the readln statement as shown below the pointer will move to the first character on the next line after reading.

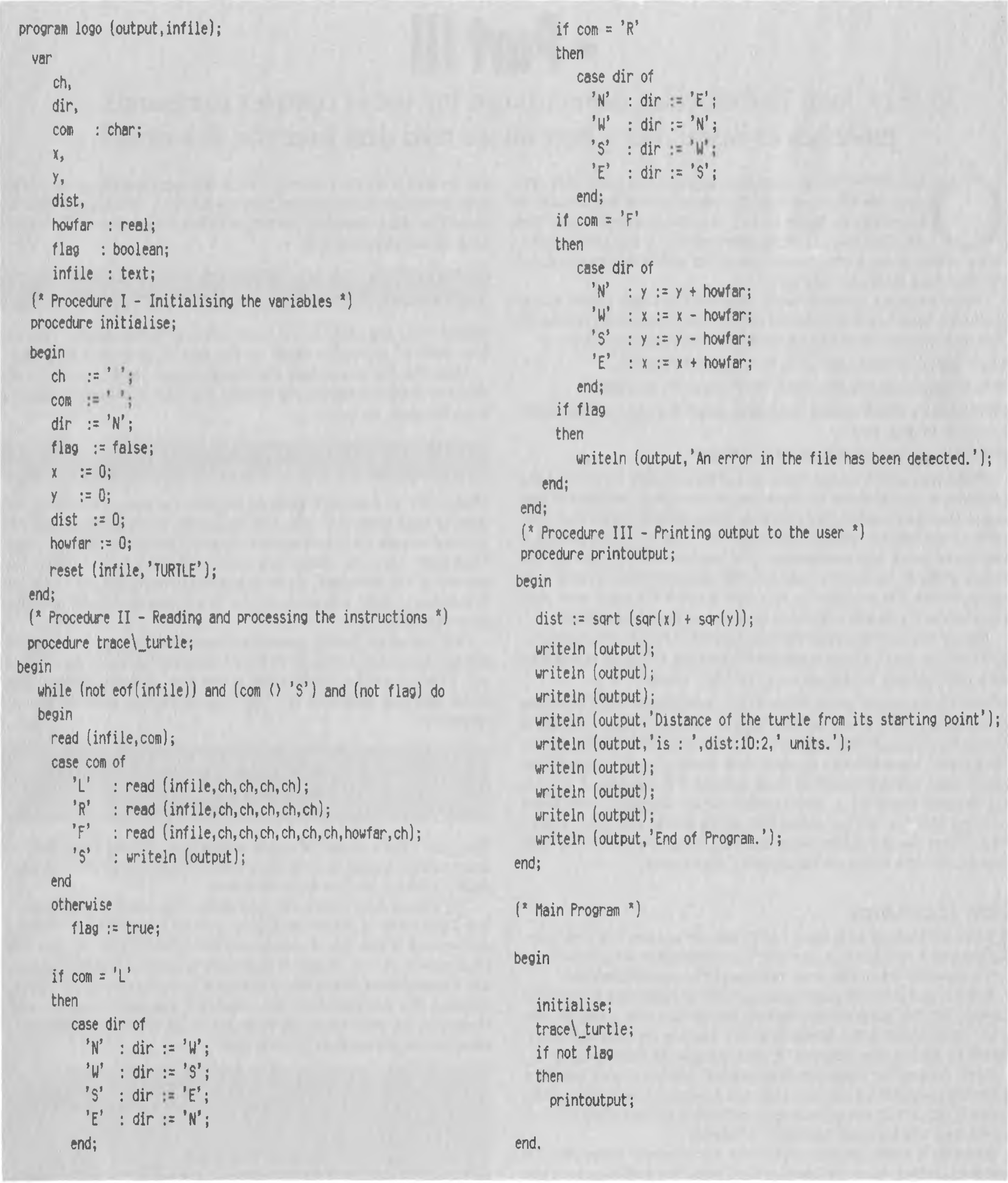
Unfortunately, Pascal generally doesn't allow you to read entire strings at a time, although some of the later versions do. Thus, if you want to read a string from a text file, such as a name, you must read one character at a time and store it in an array. For example –

```
for i=1 to name\length do
 read (infile,name\[i\]);
```

This will read a series of single characters, storing them into an array called 'name' (it is assumed that all the variables have been defined and a file has been opened)

The Pascal editor performs two tasks: first, each time you press the Return key, it places an EOLN (End Of LiNe) marker. Second, at the end of the file, it places an EOF (End Of File) marker. The importance of this, is that when reading from a file, the markers can be read in as characters. Therefore, you must be careful not to confuse the markers with the character you are trying to read. However, the markers have their use, since they can be placed in conditional statements, for example –

```
while not eof(infile) do
begin

 read (infile,num);
 total := total + num;

end;
```



```
program logo (output,infile);
 var
    ch,
    dir,
    com   : char;
    x,
    y,
    dist,
    howfar  : real;
    flag   : boolean;
    infile  : text;
 (* Procedure I - Initialising the variables *)
 procedure initialise;
 begin
    ch   := ' ';
    com  := ' ';
    dir  := 'N';
    flag  := false;
    x    := 0;
    y    := 0;
    dist  := 0;
    howfar := 0;
    reset (infile,'TURTLE');
 end;
 (* Procedure II - Reading and processing the instructions *)
 procedure trace\_turtle;
begin
   while (not eof(infile)) and (com () 'S') and (not flag) do
   begin
    read (infile,com);
    case com of
       'L'   : read (infile,ch,ch,ch,ch);
       'R'   : read (infile,ch,ch,ch,ch,ch);
       'F'   : read (infile,ch,ch,ch,ch,ch,ch,howfar,ch);
       'S'   : writeln (output);
    end
    otherwise
       flag := true;

    if com = 'L'
    then
       case dir of
         'N'  : dir := 'W';
         'W'  : dir := 'S';
         'S'  : dir := 'E';
         'E'  : dir := 'N';
       end;
    if com = 'R'
    then
       case dir of
         'N'  : dir := 'E';
         'W'  : dir := 'N';
         'S'  : dir := 'W';
         'E'  : dir := 'S';
       end;
    if com = 'F'
    then
       case dir of
         'N'  : y := y + howfar;
         'W'  : x := x - howfar;
         'S'  : y := y - howfar;
         'E'  : x := x + howfar;
       end;
    if flag
    then
       writeln (output,'An error in the file has been detected.');
  end;
 end;
 (* Procedure III - Printing output to the user *)
 procedure printoutput;
begin

   dist := sqrt (sqr(x) + sqr(y));

   writeln (output);
   writeln (output);
   writeln (output);
   writeln (output,'Distance of the turtle from its starting point');
   writeln (output,'is : ',dist:10:2,' units.');
   writeln (output);
   writeln (output);
   writeln (output);
   writeln (output,'End of Program.');
end;

(* Main Program *)

begin

   initialise;
   trace\_turtle;
   if not flag
   then
     printoutput;

end.
```

Listing 1. *This simple Pascal program simulates the Logo Turtle. It reads the instructions from a file and, after processing them, gives the displacement of the turtle.*

Here, if the file pointer is not on the EOF marker thethe condition will be true, thus a number will be read and added to the variable total.

The case statement is the equivalent of a group of 'if' statements. Its format is –

```
case num of

 1 : score := 40;
 2 : score := 30;
 3 : score := 15
 4 : begin

     score := 0;
     writeln (output,'No score!');

   end;

end
otherwise

 writeln (output,'Number out of range.);
```

This works by using the variable num (which must have a value) to decide which statement will be carried out. If num is equal to 1, the first statement will be executed (that is, score := 40); if num is equal to 2, the second statement will be executed, and so forth. Only one of the statements in the case will be executed. The case statement is usually ended with an end, however, in this case, there is an otherwise command which performs the same task as the else. If none of the conditions are met in the case, the statement(s) after the otherwise command will be performed. It is important to note that if the otherwise command was not included and the variable num was out of range (that is, not equal to 1, 2, 3 or 4) the program would crash.

The case statement is not only limited to numbers. You may also use characters and boolean variables. For example –

```
case letter of
 'a','A' : writeln (output,'Pass');
 'b','B' : writeln (output,'Fail');
```

In this case statement, if the letter is equal to either 'a'or 'A', the first statement is carried out; but if letter is equal to 'b' or 'B', the second statement is carried out.

Now we'll cover each of the procedures in Listing 1 –

Procedure initialise: In this procedure all the variables are given their initial values. Also a file called TURTLE is opened using the reset command.

Procedure trace/turtle: Here, the computer simulates the movement of the turtle. These are the variables used:

flag – if true then an error on file has been detected.

com – is the current command which has been read in.

dir – is the direction which the turtle is facing.

ch – is a dummy variable to read in extra characters.

howfar – is the number of units the turtle moves.

x and *y* – are the co-ordinates of the turtle; its starting position is (0,0).

The steps which the program takes when tracking the turtle's movement are –

1) Check three conditions: first, check the file pointer is not at the End Of the File (if it is at the EOF and you try to read in a character, then the program will crash); second, check the present command is not STOP; and third, check the flag is not set to true. If any or all of these are true, then the procedure stops.

2) Read in a letter and store it in the variable com.

3) If com is equal to L, R, F or S then execute the appropriate statement; if not, then set flag to true. If com is equal to L, it means that the current command is LEFT. Therefore, four extra characters must be read (E, F, T and a space) such that the file pointer is moved to the first letter of the next command. The same is done with the RIGHT command, although five characters must be read since the word RIGHT is one character longer than LEFT. If com is equal to F, then the command is FORWARD. The same process is carried out, although this time, the number of units to go forward must also be read. If com is equal to S, nothing has to be done, since after the completion of the loop, the while statement will not meet all the conditions, since com equals S, and the procedure will end.

4) Check whether com is equal to L. If it is, the direction of the turtle must be changed. To do this, imagine the points on a compass: if you are facing north and you turn to your left, you will be facing west.

5) Check whether com is equal to R. If it is, apply the same reasoning as in number 4 above, but this time turning right.

6) Check whether com is equal to F. If it is, then increase or decrease x or y depending on the direction which the turtle is facing. For example, if the current direction is north, then the y co-ordinate must be increased; if the direction is west, the x co-ordinate must be decreased. The x or y co-ordinates will be increased or decreased by how far (units which the turtle moved).

7) Check whether flag is equal to true. If it is, an error has occurred with the file. An error message will be then printed on the screen. Flag would be set to true, if (and only if) the command (com) read in does not equal either L, R, F or S. If this happens, you must check the file for such things as spelling mistakes or extra spaces.

8) Go to number one above and repeat the process until one of the conditions in the while statement is not met.

At the completion of this procedure, if no error has been detected, x and y will each have a value which will be used in the calculation of the turtle's displacement.

Procedure printoutput: In this procedure, the displacement of the turtle is calculated using the Pythagoras' theorem ($a^2 = x^2 + y^2$). The rest of the procedure prints out the results to the screen.

Main Program: Unlike the programs we covered in Parts 1 and 2, the main program section has an if-then statement in it. It is possible to place any statement(s) in the main program. In fact, you do not have to program using procedures at all, by writing the whole program in one block. However, a program is easier to understand and de-bug (check for errors) if it has been written in procedures. The if-then statement has been placed in the Main Program, to avoid printing out the results if an error has occurred with the data file (as explained above in the trace?turtle procedure).

That completes our introduction to Pascal – as you have probably noted it's an excellent starting point for novice programmers (as mentioned in Part 1, it was written primarily as a teaching tool). If you'd like to try programming in Pascal, the Microsoft implementation is $495 and Borland's Turbo Pascal is $210. □

CREATING WINDOWS

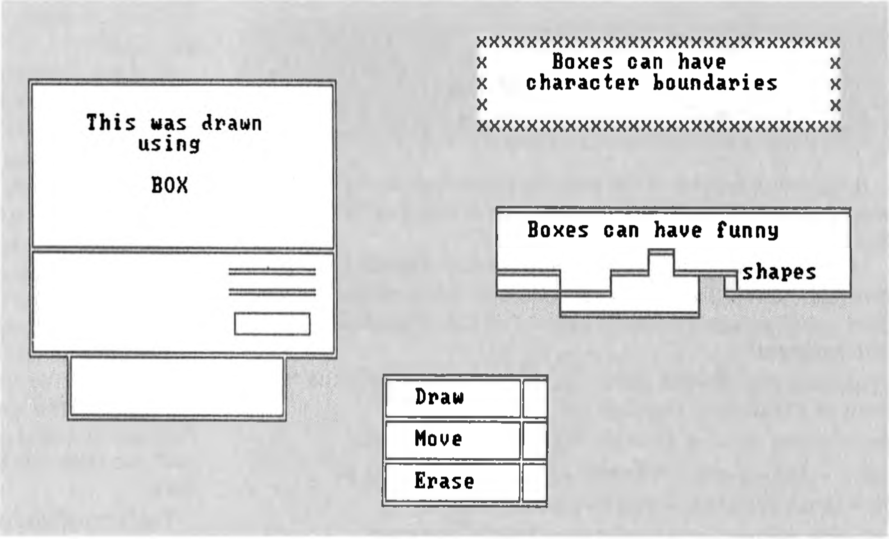
Rod Worley has found two useful public domain utilities – one for creating windows and the other for pop-up menus.

WINDOW BOSS is a library of functions to help the C programmer provide windows on the screen (CGA, mono, EGA). The source code is not provided (but can be obtained if you register). The disk contains codes compiled by the small model under the more popular compilers (C86; Lattice 2,3; MSC 3,4,5; Datalight; Turbo; Quick C), and a code compiled for use within the Quick C programming environment. This disk is basically a sample to illustrate the power of the windowing library. To use the library in a large program, it may be necessary to register and obtain the source so the code can be compiled in the larger model. There is at least one earlier version of the library in the public domain/shareware arena (PC Blue #324). This version, PC-Sig #873, supports the more recent compilers.

To help illustrate the power and speed of the library, a demonstration program is provided on the disk in both executable and source format. It is almost worth purchasing this disk just to run the demonstration program. The speed and power of the windowing library is clearly illustrated – even on my ancient 4.77MHz PC compatible. Windows flash up and vanish at incredible speed, text is written to windows and the windows scroll, also at an astonishing rate. The source code to the demonstration program is provided so you can compile and run the demonstration program to satisfy yourself; it is not all written in an assembler. I successfully compiled and ran the demonstration using Quick C and the medium memory model (the library for Quick C is compiled under the medium memory model).

The basic functions provided by the library allow the user to create a large number of pop-up windows. Windows of any size can be popped up anywhere on the screen, moved, titled and written (using a 'print-f' style of function). Windows can overlap, and any window can be brought to the top so it obscures the portions of other windows that it overlaps. The actual number of functions provided are too large to describe in detail, but the list seemed quite comprehensive. Some functions not included (as far as I could tell) were: the ability to highlight a particular line in a window, and any mouse interface. However the library does include primitive functions to set the attribute of a character, and to determine the window and current position of the cursor.

To test how easy the library was to use, I wrote a simple program using Turbo C and Window Boss to imitate a menu and help window. For the menu I created an empty surrounding window and one-line windows within the outer frame for each menu item. The selected item was made to flash by changing the attribute of the window (this can be done for an entire window, but not a single line within a larger window). The help window is filled by reading lines of text from a file. The



Boxes can have character boundaries

This was drawn using BOX

Boxes can have funny shapes

The simple WYSIWYG nature of Box is appealing for creating relatively fixed pop-up windows. It can be used to create unusual shaped windows quite easily and quickly, and the Pascal code provided makes it easy to add menus and help windows to a program.

help window also shows that the 'print-f' function handles the boarder correctly (truncation or wrap can be selected). I was pleased with how easy this exercise proved to be.

I recommend that anyone wishing to create software with pop-up windows, try this windowing library. It may be quite adequate for your needs, and even after registration it still works out quite inexpensive.

Box

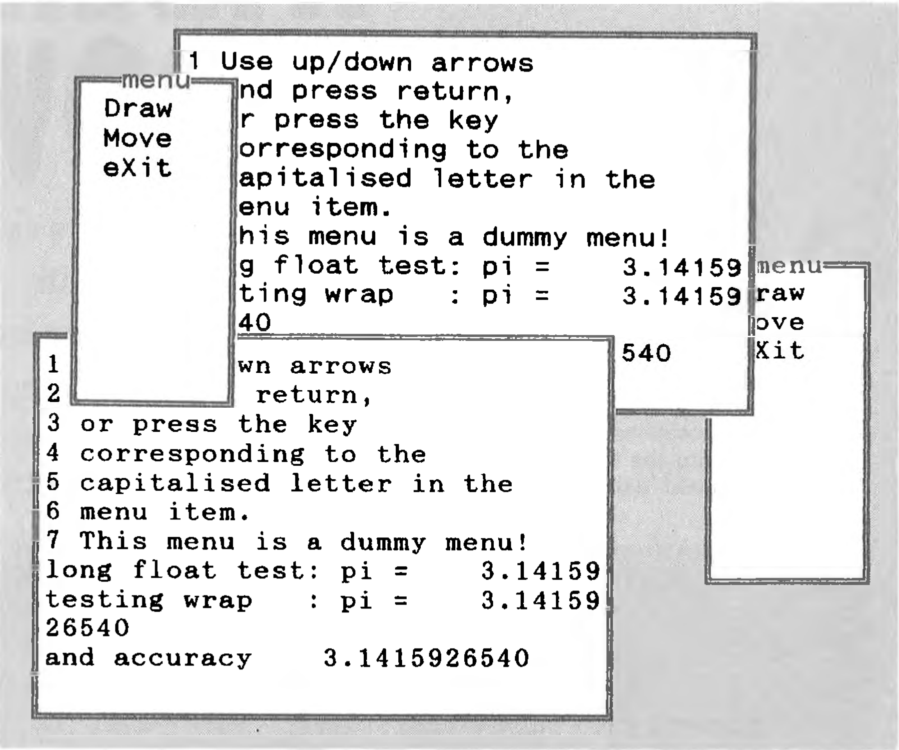
BOX, PC-SIG DISK #842, is a handy utility for creating pop-up screens. When I first tried it, I thought it paled into insignificance when compared to Window Boss. However, it is quite different and useful in its own way. In its basic use for creating screens, you first select the type of boundary you want, and move the cursor around the screen to create the window frame. When you change direction, the correct corner piece is inserted and, likewise, the correct tee or crosspiece is inserted when you meet another portion of the boundary. (There is a little difficulty in joining boundaries of different types, but it can be done – see the example.)

After drawing the boundary, the relevant text can be inserted or the interior filled with whatever colour you wish. You just 'paint' your pop-up screen, and save it to disk. You can reload it for further editing later if you wish. The program remembers the name of the last used file, which is a convenience if that is the one you wish to use. If not, then it seems necessary to backspace over the entire filename. The program uses full 25 by 80 screens, in text mode, which corresponds to files of size 4K if saved in memory image format (which is very convenient).

To use the screens in a program, the memory image file can be block read into a 25 by 80 by 2 array as a full screen image (character plus attribute for each screen position). This can then be block moved to the screen memory, possibly saving the existing screen first. Of course, only a portion of the image need actually be transferred to the screen memory if that is all that is wanted. A sample Turbo Pascal program is provided to show how to do this, giving a menu pick with highlight bars. The program is in Version 3 of Turbo Pascal, but it is easily converted to Turbo Pascal 4 by inserting the right 'uses' statements and changing the type of a record. The highlighting of the menu bars is done by changing the attributes of each character, as Box only provides the means of creating, saving and editing the screen images. It was disappointing to note that the menus in Box do not seem to have been created by Box.

The simple WYSIWYG nature of Box is appealing for creating relatively fixed pop-up windows. You can create quite unusual shaped windows (see the illustration) quite easily and quickly, and the Pascal code provided makes it easy to add menus and help windows to your program. It would be a little harder to create them with Window Boss (probably a little trial and error until things looked exactly right on the screen). However, with a number of windows, Window Boss takes care of saving what is in each window if it is overwritten, and provides scrolling and formatted printing in the windows.

As I see it, Box would be the choice if all you wanted was pop-up windows with fixed text, such as menu picks or fixed help screens, while Window Boss would be the choice if the windows were to be written to. □



```
              1 Use up/down arrows
   menu        nd press return,
   Draw        r press the key
   Move        orresponding to the
   eXit        apitalised letter in the
               enu item.
               his menu is a dummy menu!
               g float test: pi =    3.14159 menu
               ting wrap    : pi =    3.14159 raw
               40                               ove
1                                       540     Xit
2              wn arrows
3 or press the  return,
3 or press the key
4 corresponding to the
5 capitalised letter in the
6 menu item.
7 This menu is a dummy menu!
long float test: pi =    3.14159
testing wrap    : pi =    3.14159
26540
and accuracy     3.1415926540
```

The basic functions provided by the Window Boss library allow the user to create a large number of pop-up windows. Windows of any size can be popped up anywhere on the screen, moved and titled. Windows can overlap, and any window can be brought to the top so it obscures the portions of other windows that it overlaps.

AMSEC

This review was prepared for AMSEC by Rod Worley. AMSEC is an Australian software evaluation group with consultants in the workplace, in schools and in tertiary institutions around Australia. It can be contacted at PO Box 140 Hurstbridge 3099 Vic. or PO Box 1339, Armidale 2350 NSW.

Product Details

Product: Window Boss PC-Sig #873 and Boss PC-Sig #842
Distributor: Manacomm Pty Ltd
Freepost No. 29, PO Box 509
Kenmore 4069 Qld
(07) 374 1311
Price: Please call for current pricing and cost of postage, etc.
'It is almost worth purchasing this disk just to run the demonstration program.'

WIDER WINDOWS

AS PROMISED in 'Creating Windows' in November, we will now have a closer look at the ways in which pointers may be used so they are even tighter with memory. The problem, you may recall, was in the usage of Data Segment by Turbo Pascal. We solved that one neatly by shifting our screen storage locations out of the Data Segment and into the Heap. That is all very well and good, either for folks who have a lot of memory in their computers, or who do not develop or write very large programs. In any case, leaving elbow room in the 'Heap' is always handy and does no harm.

Turbo Pascal pointers can be even tighter with memory, than shown by Rod Worley in Creating Windows in November.

So, having created the dynamic variable associated with an untyped pointer last time, and moved a block of memory to it, we gazumped some 4000 bytes per screen saved. Our aim should now be to do the same, or a similar process, but to be a lit-

```
program       WiderWindcw;

Uses    Crt;

Type
   ScrnStorageArray = Array[1..2000]
                              of Byte;

   StoragePointer  = ^ScrnStorageArray;

Var
  StorePtr :  Array [1..4] of
                         StoragePointer;
  I         : integer;
  ScrnStart : word;
  Size      : integer;

{-------------------------------------}
Function    ScreenSegment : Word;

Begin
     If Mem[$0:$449] = 7  then
        ScreenSegment := $B000
     else
         ScreenSegment := $B800;
end;
{-------------------------------------}
procedure      Framewindow21;
Var
   I,X1,Y1,X2,Y2      : integer;
   base,offset,pos,C : word;
begin
     X1 := Lo(WindMin);
     Y1 := Hi(WindMin);
     X2 := Lo(WindMax);
     Y2 := Hi(WindMax);
     GOTOXY(1,1) ; Write(#213);
      GOTOXY(X2-X1+1,1) ;
      Write(#184);
      GOTOXY(1, Y2-Y1+1) ;
      Write(#192);
 For  I := 2 to (X2-X1)  do
     begin
      GOTOXY(I,1); Write(#205);
      GOTOXY(I, Y2-Y1+1);
      Write(#196);
     end;
 For  I := 2 to Y2-Y1   do
     begin
      GOTOXY(1,I); Write(#179);
      GOTOXY(X2-X1+1,I);
      Write(#179);
     end;
 Pos := (Y2 * 160) + (X2 * 2);
 Mem[ScrnStart:Pos] := ord(#217);
 Mem[ScrnStart:Pos+1] :=
            ord(Mem[ScrnStart:Pos+1]);
 end;
 {-------------------------------------}
 {-------------------------------------}
    { procedure   to Save Segment }
 {-------------------------------------}
 procedure   SavScrnSeg(X1,Y1,X2,Y2 :
                              integer;
                     SP : StoragePointer);
 Var
      I, Row, Col, C : integer;
 Begin
       I := 1 ;
 For Row := Y1 to Y2  do
  begin
   For Col := X1 to X2   do
    begin
       C := ((Row-1) * 160 ) +
                      (( Col-1) * 2);
       SP^[I] := Mem[ScrnStart: C];
       SP^[I+1] := Mem[ScrnStart:C+1];
       Inc(I,2);
    end;
  end;
end;      { end of Save Segment }
{-------------------------------------}
{-------------------------------------}
   { procedure to Restore Segment }
{-------------------------------------}
procedure   RestScrnSeg(X1,Y1,X2,Y2 :
                              integer;
                    SP : StoragePointer);
 Var
      I, Row, Col, C : integer;
Begin
     I := 1;
 For  Row  := Y1 to Y2  do
  begin
    For Col := X1 to X2 do
      begin
         C := ((Row-1) * 160) +
                       ((Col-1) * 2);
         Mem[ScrnStart:C] := SP^[I];
         Mem[ScrnStart:C+1] := SP^[I+1];
         Inc(I,2);
      end;
  end;
end;
      { End of 'RestScrnSeg'}
{-------------------------------------}
{-------------------------------------}
   { procedure to setup Window }
{-------------------------------------}
```

tle less prolific with even the Heap memory that we use.

If we analyse the situation with some degree of foresight, then we can appreciate that it is very infrequently necessary to allocate more than 2000 bytes (half of the screen) to any given window, and indeed the vast majority of windows in use for menus, error messages and the like will use even less than that, typically 1000 bytes. So, at this point we might surmise that a simple one-liner, of the form :

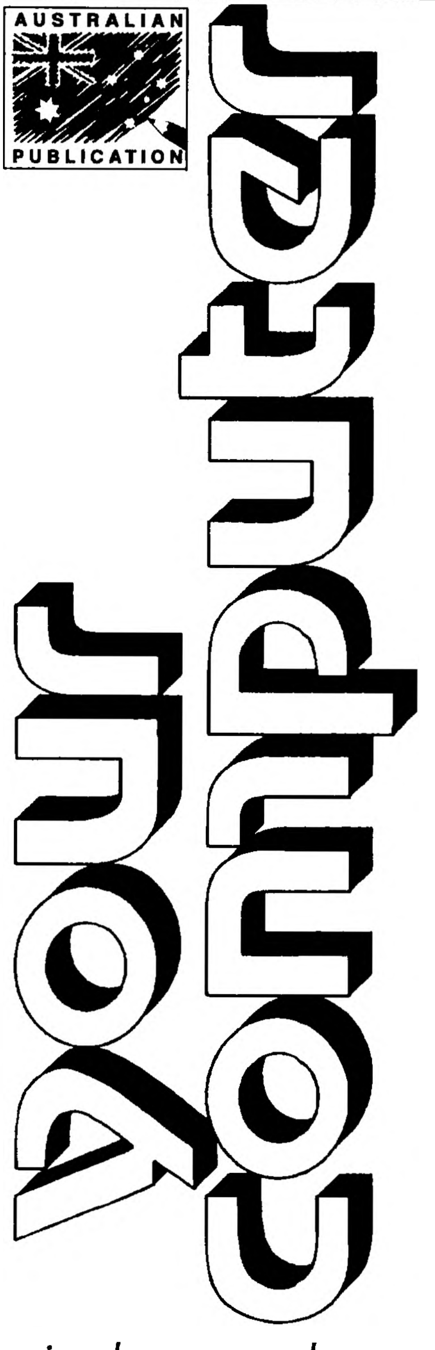
```
Move(Ptr(ScrnSeg:0)^,SP^,2000)
```

– is all that is needed to achieve the reduction, by at least 50 per cent of our memory usage in the Heap. Unfortunately, not so. The 'Move' procedure in Turbo Pascal works on contiguous bytes of memory, and even though we know where the window will start, the memory overwritten by the window is not in contiguous bytes. It is, in fact, a chunk of bytes (the window width), followed by some 'other screen bytes', then some more window bytes; and so on. Alas, slight complexity is forced upon us.

The procedures and short program in Listing 1 start with a type declaration for an important data structure, and for the type of pointer that we will later use as a variable to point to that structure. As you can see, the 'ScrnStorageArray = Array [1..2000] of Byte' line, is the half screen I

```
procedure  PutWindow(X1,Y1,X2,Y2 :
                               integer;
                      Text, Back : Word ;
                    SP : StoragePointer);
Begin
    SavScrnSeg(X1,Y1,X2,Y2,SP);
    Window(X1,Y1,X2,Y2);
    TextBackGround(Back);
    TextColor(Text);
    ClrScr;
    FrameWindow21;
    GOTOXY(2,2);
    Write('Memory - ',MaxAvail);
    Delay(500);
end;      { of 'PutWindow' }
{-------------------------------------}
     { Start of main program }
{-------------------------------------}
Begin
     ClrScr;
     ScrnStart := ScreenSegment;
     TextColor(White);
     TextBackGround(Blue);

   { Create some background text }
{-------------------------------------}
  For i :=  1 to 95 do
    Write('This sentence on screen.');
{-------------------------------------}

  { Allocate space to store screen, }
  { and then save it to that space. }
  { Same thing, four times.         }
{-------------------------------------}
  GetMem(StorePtr[1],2000);
  PutWindow(5,10,75,23,Black,Lightgray,
                         StorePtr[1]);
  Delay(1500);

  GetMem(StorePtr[2],910);
  PutWindow(30,5,70,15,Black,Cyan,
                         StorePtr[2]);
  Delay(1500);

  GetMem(StorePtr[3],800);
  PutWindow(40,15,79,24,Black,Green,
                         StorePtr[3]);
  Delay(1500);

  GetMem(StorePtr[4],300);
  PutWindow(2,2,30,7,White,Blue,
                         StorePtr[4]);
  Delay(1500);

  RestScrnSeg(2,2,30,7,StorePtr[4]);
  FreeMem(StorePtr[4],300);
  Delay(1500);

  RestScrnSeg(40,15,79,24,StorePtr[3]);
  FreeMem(StorePtr[3],800);
  Delay(1500);

  RestScrnSeg(30,5,70,15,StorePtr[2]);
  FreeMem(StorePtr[2],910);
  Delay(1500);

  RestScrnSeg(5,10,75,23,StorePtr[1]);
  FreeMem(StorePtr[1],2000);
  Delay(1500);
End.
```

Listing 1. WiderWindow uses pointer techniques to reduce Heap usage for screen storage. No data segment is used for screen storage and only those segments of screen that are overwritten are stored.



was referring to when I said that we would seldom have to use a window this large. Throughout this exercise I will assume that all windows opened have this maximum value. In reality, you may have the type of array as small, or as large, as you choose (up to 4000 bytes being the practical limit). However, given the speed of the restoration routine being used, on a slow (4.77MHz), XT type machine, I would not exceed the 2K limit if I were you. With windows that are larger, use 'Move' to shift the whole screen, as we did previously. On 8, 10 and 12 MHz machines, the relative slowness of these routines is not noticeable – they operate in the proverbial 'blink of an eye'.

So, what have we achieved so far? Well, the data structure is ready to go, and the pointer to that array is established. Now, to the variable declarations. 'StorePtr : Array [1..4] of StoragePointer' establishes the small array of pointers that we require. Referring to them by subscript means that we will later be able to call them in the appropriate order with less chance of confusion than if they all had different names.

The 'SavScrnSeg' procedure comes next. This is plainly a simple nested 'for' loop, in which we are indexing the columns across the window before changing the row (Y-value). What is the 'C' variable, local to this procedure? It is the calculated offset of the particular row/column set we are attempting to save at this instant. The local variable 'SP' is of the 'StoragePointer' type and we are going to pass it the value of one of the pointers in the 'StorePtr' Array (for example, StorePtr[1]).

Having calculated this offset value, we need to shift the contents of that one location on screen, and its attribute byte to our Array of 2000 bytes, somewhere in the Heap. This is achieved on a byte by byte basis thus:

```
SP^[I]    := Mem[ScrnStart : C ]  ;

SP^[I+1]  := Mem[ScrnStart : C+1] ;
```

This is a method of indexing within the 'ScrnStorageArray' using the StorePtr acting locally here under the pseudonym 'SP', and 'SP' with the caret between the pointer identifier and the [index] tells the program which dynamic array we mean. 'Mem' is of course the standard Turbo Array which allows us access to any part of memory.

If, later in the program, we were to declare a 'New(StorePtr[1])', then the whole of the 2000 bytes associated with that, and successive pointers of the type, would be allocated each time a 'New' procedure was activated.

This means 8K of Heap for the four windows of this program. Fine! That is twice as good as 16K out in the Heap as we had before. However, we can go one better still. Since it is likely that the size of a window for an error message, menu, and so on, will be determined pretty much in advance by the nature of the application being written – it is not necessary to declare the full 2000 bytes every time, and we can use a statement of the form:

```
GetMem(StorePtr[1], Size)  ;
```

– where the size of the dynamic variable equals the window's inclusive area multiplied by two. Why not avoid the array structure altogether? Well, simply because we need a structured variable which we can access in a known, sequential, and precise way, to store and replace all of those non-contiguous bytes of screen memory. Using both of these techniques we get dynamic allocation, data segment freedom and minimum Heap usage. The penalty is a loss of speed, but I don't think it is objectionable, up to about 2K window size, as mentioned previously.

Keeping track of the windows open is no problem, if you are working with five, four, or less windows.

PutWindow

'PUTWINDOW' PROCEDURE CALLS follow each 'GetMem' statement, and this procedure does a couple of obvious and important things for us. Firstly, it calls 'SavScrnSeg' with the co-ordinates of the window being opened, and passes the value of the current pointer to that procedure. The rest of the routine opens the window and, in this example, prints the current value of 'MemAvail'. The last couple of lines effectively track the memory usage for us.

The last real 'work horse' routine is the 'RestScrnSeg' procedure. It is, in essence, the exact reverse of 'SavScrnSeg', writing each byte of the 'pointed to' array back into the 'Mem' Array segment we have specified. Note the use of 'Inc(I,2)', instead of 'I := I + 2' – this gives a slight gain in speed where it is important, since the 'Inc' procedure generates optimal compiled code.

Having used 'RestScrnSeg' to put everything back where it came from, we must not forget to return the Heap to its original state. Hence, we need:

```
FreeMem(StorePtr[3], 800)  ;
```

– or something similar, after each screen restoration procedure.

Beware!!

YOU MUST USE 'RestScrnSeg' and 'FreeMem' in exactly the opposite order to that in which they were created originally. If not, several windows will wink out simultaneously, and intermediate pointers with their associated memory will not be properly de-allocated. In some cases, it may be a final menu choice, or some such definite condition, and then it may be acceptable for all windows to 'pop-off'. If you wish to back track through established windows, then be strict in your observance of this de-allocation order.

Keeping track of the windows open is no problem, if you are working with five, four, or less windows. A more complex system will require a table of pointers and window co-ordinates, or a linked list of pointers and windows.

Some last details – the example program has delays inserted so you can follow the action. The framing procedure has a double bar at the top, and is otherwise a single frame, and a variable called 'ScrnStart' is calculated for later use in procedures since this is faster than calling the function 'ScreenSegment' repeatedly in the 'SavScrnSeg' and 'RestScrnSeg' procedures. I have spread out the 'Save' and 'Restore' usages in the main program so that the sequence of usage is obvious. There is nothing to stop you from integrating all window data, as alluded to previously, into a single table when the usage is well understood. So – how did we end up, memory-wise? A quick review of this example shows a Heap usage of 4010 bytes for four windows, of various sizes, from small to large. This gives us a fourfold improvement and that *is* worthwhile! Happy coding. □

CHEAP WINDOWS IN PASCAL

Paul Frankel updates windows in Pascal and shows how to get the Save/Restore routines to calculate memory requirements and manage the heap.

GARY JACOBSON in *Your Computer*, November and December '89, wrote two helpful articles on making windows in Pascal. The latter article showed vast improvements in memory usage but had four limitations: there were complex calculations in nested loops; the user was responsible for calculations of memory requirements; the user was responsible for heap management; and the user was also responsible for closing windows in a strict order. This article will show how to eliminate these limitations.

The rationale behind this is: any 'speed freak' will tell you that nested loops are the basis of 'delay' routines and calculation in them obviously compounds this problem; and, novice users – experienced ones, too – are bound to make mistakes in calculating memory requirements.

Why not a procedure with a 'pass by reference' parameter?

Pascal's view of your computer's memory is that there are four segments of 64Kb maximum, namely data, code, stack and the heap. Global variables reside in the data segment, whereas Local variables and function and procedure calls are placed on the stack. This is why Local and Global variables with the same name don't interfere with each other. A return from a function/procedure call, clears the appropriate stack space, hence the scope of local variables is limited. 'Call by reference' parameters affect the data segment, whereas 'call by value' parameters simply copy the value onto the stack, and play around with that.

Our aim is to use minimum heap space to save the screens, which will retain the maximum available for other uses.

A request for a window implies writing to the screen followed by a restoration of the original material, one would presume at the same location. Only one variable, the address of the stored information is needed by the restoring routine, which suggests a function should be used. Why not a procedure with a 'pass by reference' parameter? RestoreWindow will show you how.

The algorithm

IF THE CO-ORDINATES are okay and memory is available then the algorithm for this is: save co-ordinates; increment pointer; calculate number-of-lines; calculate number-of-characters; for number-of-lines do: save number-of-characters, increment pointer; else return NIL pointer.

The user will need to take the responsibility for something, and error checking for the NIL pointer should be incorporated into the code.

Function FindVideoType (Listing 1) is global and it is used in both SaveWindow and RestoreWindow to determine the screen address. Now examine the code in Listing 2 for the SaveWindow function . . . Because the function returns a value it must be given a type – the standard type pointer will do.

SaveWindow accepts four parameters – topLeftX, topLeftY, bottomRightX and bottomRightY. It does a simple check to ensure the box shape has positive dimensions then calculates the amount of heap space required Recall that an error in either of these aspects causes the NIL pointer to be returned.

The four-byte local array WindArr is only used to simplify the saving of the co-ordinates in the Move command, and incrementing the pointer.

```
FUNCTION findVideoType: longInt;
const mono   = $B000;
    colour = $B800;
begin
  if mem[$0:449] = 7
    then findVideoType:= mono
    else findVideoType:= colour;
end;
{------------------------------------------------------------------}
```

Listing 1. Function FindVideoType is global and it is used in both SaveWindow and RestoreWindow to determine the screen address.

The Move command –

```
— move (windArr, windPtr^, sizeof(windArr)); —
```

– itself has some interesting points to note. First, the SIZEOF command is used rather than the number four; the reference manual suggests this usage for clarity and safety. Second, the name of the array, WindArr, is itself a pointer to the information held in the array. Third, WindPtr is a memory location that holds the memory location of where to store the information. As an analogy for this, say you call in at a service station and ask for directions to a certain address. The proprietor (a pointer) tells you where to go. He, she or it has no knowledge of what you want to do at that address.

The use of the caret symbol tells the compiler to use the address at which the information is to be placed.

Global variables reside in the data segment, whereas Local variables and function and procedure calls are placed on the stack.

Typecasting allows traversal of the correct number of bytes through the allocated memory, such as pointer (longInt (WindPtr)). LongInt converts WindPtr from pointer type to longInt, which enables the addition of the appropriate number of bytes. Pointer (----) now reconverts the new longInt value to a pointer type.

The predefined array variable, MEM is used to directly access one byte anywhere in memory, and INC will increment a variable by the amount indicated.

WILL THE GREAT GURU ANSWER MY QUESTIO..?
SORRY, NOT TODAY, THE COMPUTER BROKE DOWN.

Pascal uses screen co-ordinates 1, 1, 80, 25 so the screen offset should compensate for that. Note also the doubling, to allow for the attribute byte. Once the offset is calculated, you simply move the appropriate number of characters to the heap, then drop down one screen line and repeat. That is, 80 character and 80 attributes bytes hence, Inc (screenOfs, 160); the last thing a function should do is to assign to the function name a value to be returned. WindPtr was modified throughout the function, so a temporary variable, WPtr, is needed to store the initial pointer position.

Summarising

THE MACHINE has done all the calculations. The window obtains its own memory and stores its co-ordinates for restoration at a later date. Minimum heap space has been used and maintained with no user intervention. Nested loops do not exist.

Restoring the window offers no great challenge now, and retrieval can be in any order (it does have some uses).

RestoreWindow (Listing 3) accepts two parameters, a Boolean –

```
FUNCTION SaveWindow (TopLeftX, TopleftY,
            BottomRightX, BottomRightY: byte) : pointer;
var
  windArr          : array [0..3] of byte;
  windptr, Wptr      : pointer;
  numLines, NumChars, i: Integer;
  screenSeg, screenOfs : longI nt;

procedure error (i: byte);
const NoWindow = 'SaveWindow could not allocate enough Memory';
   Co_ords  = 'Error in Coordinate order';
begin
 if i = 0 then writeln (Co_ords) else writeln (NoWindow);
 saveWindow:= NIL;
 halt;
end;
begin
 if BottomRightY >= TopleftY
   then numLines:= BottomRightY-TopleftY+1 else error (0);
 if BottomRightX >= TopLeftX
   then NumChars:= (BottomRightX-TopLeftX+1)*2 else error (0);
 i:= NumLines * NumChars + sizeof (windArr);
 getmem (windPtr, i);
 if windPtr = NIL then error (1)
 else begin
   WPtr:= windPtr;
   screenSeg:= findVideoType;
   screenOfs:= (TopleftY-1) * 160 + (TopLeftX-1)*2;
   windArr[0]:= TopLeftX;
   windArr[1]:= TopleftY;
   windArr[2]:= BottomRightX;
   windArr[3]:= BottomRightY;
   move (windArr, windPtr^, sizeof(windArr));
   windptr:= pointer (longInt (windPtr) + sizeof (windArr));
   for i:= 1 to numLines do begin
     move (mem [screenSeg:screenOfs], windPtr^, NumChars);
     windptr:= pointer( longInt( windPtr) + NumChars);
     inc (screenOfs, 160);
   end;
   saveWindow:= WPtr;
 end;
end;
{----------------------------------------------------------------}
```

Listing 2. *This is the code for the SaveWindow function.*

if you think you may want to restore the same window many times – and a 'pass by reference' pointer to free the heap space used. (Set up the windows you want for multiple accesses first, to help prevent heap fragmentation.)

If the pointer exists then; retrieve co-ordinates; increment pointer; calculate number-of-lines; calculate characters-saved; for number-of-lines do restore characters-saved increment pointer; free memory.

Looking at the code, notice a check that the referenced pointer contains appropriate data before bursting into action. Retrieval of the window co-ordinates allows the calculation of the screen offset and the number of characters to move to each line.

The GotoXY simply positions the cursor at the last position of the restored window – it had to go somewhere!

Finally, free the heap memory and return a NIL pointer.

```
PROCEDURE RestoreWindow (VAR WindPtr: pointer; loselt: boolean);
var
  windArr          : array [0..3] of byte;
  Wptr             : pointer;
  numLines, NumChars, i: integer;
  screenSeg, screenOfs : longInt;

begin
 if windPtr <> NIL then begin
  wPtr:= windPtr;
  move (windPtr^, windArr, sizeof(windArr));
  numLines:= WindArr[3] - windArr[1] + 1;
  NumChars:= (WindArr[2] - windArr[0] + 1) * 2;
  gotoxy (WindArr[2], WindArr[3]);
  screenSeg:= findVideoType;
  screenOfs:= (windArr[1]-1) * 160 + (windArr[0]-1) * 2;
  windptr:= pointer( longInt(windPtr) + sizeof(windArr));
  for i:= 1 to numLines do begin
   move (windPtr^, mem [screenSeg:screenOfs], NumChars);
   windptr:= pointer( longInt( windPtr) + NumChars);
   inc (screenOfs, 160);
  end;
  if loselt then begin
   i:= NumLines * NumChars + sizeof (windArr);
   freemem (WPtr, i);
   WindPtr:= NIL;
  end else windPtr:= Wptr;
 end;
end;
{------------------------------------------------------------------}


    Now some simple code to test it .....

program Pascal_Windows;
uses crt;
var
  scr1, scr2, scr3: pointer;
  i,j            : integer;


procedure wait_clear;
begin
 readln;
 clrscr;
end;
    {---------------- main ------------------------}
begin
 textcolor (yellow);              { just to show colour }
 textbackground (red);            { poses no problems }
 clrscr;
 For i:= 1 to 24 do begin
     Write ('a screen full of rubbish');
     Writeln (' helps the windows work well');
 end;
 Scr1 := SaveWindow (18, 20, 80, 20);  { a line of screen data }
 Scr2 := SaveWindow (10, 10, 24, 15);  { now a block }
 Scr3 := SaveWindow (15,  1, 15, 20);  { how about a column }
 wait_clear;
 restoreWindow (Scr2, True);          { any order will do }
 wait_clear;
 restoreWindow (Scr3, False);
 wait_clear;
 restoreWindow (Scr1, True);
 wait_clear;
 restoreWindow (Scr3, True);          { got it again }
 wait_clear;
 restoreWindow (Scr2, True);          { nothing! - already LostIt}
 readln;
end.
```

Listing 3. RestoreWindow accepts two parameters, a Boolean and a 'pass by reference' pointer to free the heap space used.

In a second article, we'll cover 'boxing' in the window, a complex Array/Menu Selector and the TPU file on disk, for the asking. □

BORDERED WINDOWS

In August, Philip Frankel showed how easy it is to manage windows in Turbo Pascal. Now, he discusses how to border the windows and covers the concept Achoice().

IN AUGUST'S 'CHEAP windows in Pascal', I demonstrated how to save and restore windows using Turbo Pascal with minimum heap space and ease of management. A function called FindVideoType was developed, which located the screen address for colour or monochrome monitors, and all that remained was to find the offset to enable a screen save or restore.

The predefined array MEM which allowed direct memory addressing was also examined. Here, we will put a border around the windowed area and cover the concept and use of the function Achoice().

TextAttr is a predefined variable of type Byte, which holds the currently selected text attributes as follows:

```
4 bits - foreground colours --> 16 possible colour choices.
3 bits - background colours --> 8 possible colour choices.
1 bit  - blink or not
```

Instead of declaring a new variable for colour we may as well use this one. Calls to the standard procedures TextColour() and TextBackground() actually modify the text attribute byte like this:

```
TextAttr = textbackground * 16 + textcolour
```

For example –

```
TextAttr:= blue * 16 + yellow + blink
```

Frequently, you may want to put a border around things which aren't actually windows, so to avoid excess code, function SaveWindow() deliberately did not contain the code to make the border. Listing 1 shows the 'box' procedure which draws the border. It accepts six parameters: TopLeftX, TopLeftY, BottomRightX, BottomRightY, BorderType and BorderColour. As none of these values exceed 255, we can use type Byte. The border style can be in the range 1 . . 3, anything else causes no border to be displayed.

```
1 --> single lines
2 --> double lines
3 --> combination, and of course the reader could develop more.
```

The border is only written once per window, hence standard procedures are fast enough and easy to understand, so use them.

First, save the old attribute byte to enable restoration of the original colours, then set up the border colour and choose which character set to display. The contents of the window may be simple data, a menu or an array of data. Obviously the contents of an array would require many writes to the screen as you move through the array. Current trends in computer development are aimed at increasing speed, so programming should reflect this.

Listing 2 and 3 perform exactly the same function, except that the FastWrite procedure can be much faster. On 50,000 repetitions, FastWrite proved to be 39 times faster in doing the task. When 1000 repetitions were tested, FastWrite was down to twice as fast. If you have 500 or fewer items to display, there is probably no speed advantage to be gained. However, in a windowed area, Write produces an unacceptable effect.

Write automatically moves the cursor to, and clears, the next usable position after displaying each character. If the word to be written, truncated if necessary, is the width of the window, write(-ing) will wipe out the right hand border on that line. One solution is to finish writing one space before the border, resulting in a

```
PROCEDURE BOX  (TopLeftX, TopLeftY, BottomRightX, BottomRightY,
            BorderType, BorderColour: Byte);
CONST
{     Lt1 represents left top - character set 1, and so on.        }
 Lt1   = 218;    Rt1   = 191;    Lb1   = 192;    Rb1   = 217;
 Lt2   = 201;    Rt2   = 187;    Lb2   = 200;    Rb2   = 188;
 Lt3   = 213;    Rt3   = 184;    Lb3   = 212;    Rb3   = 190;
 Horiz1 = 196;   Vert1 = 179;    Horiz2 = 205;    Vert2 = 186;

Var OldColours: byte;
PROCEDURE FRAME (Rt,Lt, Rb,Lb, Horiz, Vert: byte);
VAR I: byte;
BEGIN
  GotoXY (TopLeftX, TopLeftY);
  Write (CHR (Lt));
  For I:= TopLeftX +1 TO BottomRightX -1 DO
    Write (CHR (Horiz));
  Write (CHR (Rt));
  For I:= TopLeftY+1 To BottomRightY -1 Do begin
    GotoXY (TopLeftX,I); Write (CHR (Vert));
    GotoXY (BottomRightX,I); Write (CHR (Vert));
  END;
  GotoXY (TopLeftX, BottomRightY);
  Write (CHR (Lb));
  FOR I:= TopLeftX+1 TO BottomRightX-1 DO
    Write (CHR (Horiz));
  Write (CHR(Rb));
END;
BEGIN
 OldColour:= TextAttr;
 TextAttr:= BorderColour;
 case BorderType of
  1: Frame (Rt1,Lt1, Rb1,Lb1, Horiz1, Vert1);
  2: Frame (Rt2,Lt2, Rb2,Lb2, Horiz2, Vert2);
  3: Frame (Rt3,Lt3, Rb3,Lb3, Horiz2, Vert1);
  else
 end;
 TextAttr:= OldColours;
END;
```

Listing 1. *The 'box' procedure – refer to the text discussion.*

```
LISTING 2

Procedure WriteIt (x, y: Byte;  Astring: String);
Begin
  GotoXY (x, y);
  Write (Astring);
end;

LISTING 3

Procedure FastWrite (x, y: byte;  Astring: String);
{ ScreenAddr was found previously using function FindVideoType }
Var i, len : byte;
   ScreenOfs : word;
Begin
  len := length (Astring);
  ScreenOfs := (y - 1) * 160 + (x -1) * 2;
  For i := 1 to len do Begin
       Mem [ScreenAddr : ScreenOfs]:= ord (Astring[i]);
       Mem [ScreenAddr : ScreenOfs + 1]:= TextAttr;
       inc (ScreenOfs, 2);
  end;
end;
```

***Listing 2.** and **Listing 3.** Both listings perform the same function, but the FastWrite procedure can be much faster.*

blank column on the right hand side of the window.

On examining the FastWrite procedure, you will see that writing to the screen involves the use of the predefined array MEM, which eliminates the above problem. The use of MEM allows direct accessing of any byte in memory. It expects a variable of type Byte to be assigned to that spot, so the code has to typecast the character to be written, hence the use of the predefined function Ord().

Should the contents of the window be an array of elements from a database of people's names, you may want to be able to move forward, back or simply jump to a certain position. If that is the case, then Achoice() is the answer.

Achoice() uses a modified version of FastWrite where 'len' (see Listing 3) is the length of the string to be written, or the maximum width of the window, whichever is smaller. The algorithm is –

```
Find Screen_Address and Screen_Offset
Set_Up_Colours
While More_Array_Elements and Window_Space_Is_Available
    Call FastWrite
    Increment Screen_Offset
Wait_for_keystroke
```

Before examining function Achoice(), a study of pascal strings is in order.

The Pascal assignment statement   Astring = ''. does not actually blank out the string, it merely sets the length byte (Astring[0]) to zero and leaves the rest untouched. Declaring an array of string would set aside enough contiguous (side by side) bytes of memory to accept all data. Logically, when each element has been assigned to the array, it could be visualised to be something like this . . .

```
3THE*#-3?B=5HELLOgEI*-0*(YZ) ... etc.     ( using string[10] )
 |__________|__________|__________
```

Function Achoice() expects all unused spaces to be filled with the null character, hence procedure Afill() has been provided for this purpose. Note the use of the standard function SizeOf(), to be sure the correct length of the array is sent to Afill(). The syntax is –

```
Afill(ArrayName, SizeOf(ArrayName));
```

Failure to use Afill() before using Achoice() will give you a window full of garbage or worse.

Achoice() accepts 9 parameters: the window parameters per the Box function, colour combinations for the window text, Highlight bar and BorderColour, and the Name of the array to be displayed. The syntax is –

```
Achoice (x1,y1, x2,y2, BorderType, BorderCol, TextCol, HighCol, ArrayName);
```

Be sure to allow for the loss of two columns and two rows in your window dimensions to allow for the border. Now, blank the array using Afill() – this is crucial! Assign elements as you want them, starting with the first element. ArrayName [0] must *not* be used for data as it is used for housekeeping by Achoice() (see Listing 4). It doesn't matter if you don't use all the elements that you asked for in the variable declaration, provided you have used Afill() as shown in Listing 4.

A call to Achoice() will display as much of the array as the window coordinates will allow, and accepts any of the following keys: Home, End, PgUp, PgDn, Up/down arrows, Enter, Esc, and the first letter (case insensitive) of any element.

```
{ some code to test the use of Achoice() }
uses windows;
Const EscPressed = -1;
    MaxSize    = 100;
Var MyArr : Array [0  ..  MaxSize] of String [20];
   Nmbr: Integer;
   BorderCol, TextCol, HighCol: Byte;
Begin
  BorderCol:= blue * 16 + white;
  TextCol  := white * 16 + red;
  HighCol  := white * 16 + yellow;
  Afill (MyArr, Sizeof (MyArr));    { — THIS IS CRUCIAL — }
  MyArr [1]:= 'Hello';            { — DO NOT USE MyArr [0] — }
  MyArr [2]:= 'There';
  {       etc       }

  Nmbr:= Achoice (30,10, 45,20, 3, BorderCol, TextCol, HighCol, MyArr);
  If Nmbr > EscPressed then        {  test for Abort Option  }
    writeln (MyArr [Nmbr]);
end;
```

***Listing 4.** Code to test the use of Achoice() – refer to the text discussion.*

Pressing Enter will cause Achoice() to restore the screen and return an integer value representing the position of the array element highlighted (starting from 1, remember).

Esc will restore the screen too, but it is the Abort Option – returning a value of -1. The user should test for this in the code (see Listing 4).

Your Computer has a TPU file containing all that has been discussed in these two articles. There is also a windows.doc file, which gives a brief explanation of the functions and procedures – some of which have not been mentioned. It is available be sending a formatted floppy to the address on the contents, *enclosing return postage*. If you find a use for it, all I ask is that you acknowledge the author. □

Assembling QuickBasic... - Part 1

Jeff Richards starts a new series on Assembly language routines for QuickBasic with an overview of language routines.

ASSEMBLY language routines for QuickBasic can add useful additional facilities to the language, and do not require a detailed knowledge of Assembly language programming. QuickBasic Revision 4 incorporates a significant number of improvements and enhancements over previous revisions. One of the most useful is the facility available for building and using libraries of Assembly language routines. These user-libraries can provide both language extensions and code simplification, and the Assembly language code involved is not necessarily complex (see Jeff Richards article 'Making QuickBasic Libraries' in *YC*, Sept. '88).

For many Basic programmers, Assembly language modules are the last thing they would consider for their program – after all, everyone knows that Assembly language is tedious, complicated and error-prone.

Or is it? Assembly language can certainly be difficult to use for large comprehensive programs, but for small, special-purpose modules, it can be simple to use and easy to debug. And it certainly creates efficient code.

In order to implement the routines, an assembler that can produce Microsoft standard object (.OBJ) files is required. MASM Rev 5 is preferred, as it implements a set of compiler directives that simplify the segment definition and segment ordering tasks that have made libraries so difficult to build in the past. However, compatible assemblers are available from a number of sources, including the Public Domain.

The process of constructing an Assembly language routine can be broken down into three separate tasks, and these tasks can be tackled one at a time. The parts of the problem can be defined as –

1) The definitions and commands that the assembler needs in order to know that it is dealing with a QuickBasic routine;

2) The procedures within the routine involved with interfacing to QuickBasic;

3) The code for the actual procedure of the routine.

Assembler directives

ASSEMBLY language modules that are to become part of a QuickBasic user library must follow certain standards if QuickBasic is going to be able to find and use the library routines. These rules are –

1) The module must be defined as type FAR. This means that a 4-byte address (segment/offset) will be used to call the routine, and the routine will return to QuickBasic by popping a 4 byte return address off the stack. In MASM Rev 5, the compiler directive .MODEL MEDIUM ensures the correct definition of the module type;

2) The segments – code, data, constants and so on – must follow the correct ordering. The compiler directive DOSSEG forces the correct ordering of segments;

3) QuickBasic must be able to find the position in the routine at which to start processing. These positions are known as entry points. To advise QuickBasic of the entry points, they must be declared with the label PUBLIC.

Additional Development

ADD TO the file BITS.ASM a matching routine called BITCLR that clears a nominated bit, rather than setting it. Note that forcing a bit to zero can use the same mask as forcing it to 1, except that the mask operation is a NAND (or NOT-AND) instead of an OR. The 80x86 processor does not support a NAND opcode.

QuickBasic Interface

THE QUICKBASIC interface itself breaks down into two parts – the behaviour of the routine (what it can and can't do), and the way data is passed back and forth. The routine must follow certain rules of behaviour in order to ensure that it can co-exist happily with QuickBasic.

Firstly, ***under no circumstances must CPU register BP be altered***. This register is used by QuickBasic to provide a 'frame of reference' for its data structures. However, it is convenient to use BP for a similar purpose within the routine, so a typical Assembly language procedure will start by saving the current contents of BP on the stack, from where they can be safely retrieved immediately before the routine terminates. Therefore, the first statement in most Assembly language routines is –

```
PUSH BP
```

When the routine terminates, it is important that other registers that QuickBasic relies on have not been altered. These are SI, DI, SS and DS. If any of these registers could be altered then they must be pushed onto the stack at the start of the routine and retrieved before returning. (SI and DI are currently optional, but saving and restoring them ensures compatibility with later QB implementations.)

The last aspect of the behaviour of the routine is the way it terminates. Firstly, if any of the reserved registers have been pushed onto the stack, they must be popped off again. Then register BP must be popped off the stack.

To return to QuickBasic, a far return is executed with a parameter that refers to the number of data items that were passed to the routine when it was called. This enables the return statement to clean up the stack area into which these parameters were loaded – just to keep everything neat and tidy. Therefore the last two statements in most Assembly language routines are

```
POP BP
RET n
```

where *n* is a number that we will consider in a moment.

The second aspect of the interface of the routine to QuickBasic involves the sharing of data between the two. Most (but not all) routines will expect to receive some data from the calling program, and many will need to return some data.

All data that is passed to the routine is passed on the stack – before QuickBasic calls the routine it PUSHes the specified parameters onto the stack. The routine cannot simply POP these values off the stack, because that would destroy the return address that it needs to get back into the calling program. Instead, the value of the stack pointer (SP) is loaded into register BP, and then a knowledge of the stack structure is used to directly access these data items. It sounds complex, but in practice it is simple.

We know that the calling program has pushed the required data onto the stack. We also know that it has pushed a 4-byte return address after the data, and that the first thing our routine did was to push two more bytes – the contents of the BP register. The SP register currently points the the top of the stack, so the data the routine needs will be found on the stack at position SP+6 and onwards. We have promised not to muck around with SP (and besides, there are severe limits on how SP can be used in instructions), so to gain access to the parameters we will load the value of SP into BP and use an expression such as BP+6 to access the data.

> *Assembly language can certainly be difficult to use for large comprehensive programs, but for small, special-purpose modules, it can be simple to use and easy to debug.*

Because of the way QuickBasic pushes data onto the stack, BP+6 actually contains the last argument in the list, BP+8 has the second-last, and so on. Also note that all this assumes that the routine is a SUB, not a FUNCTION – slightly different rules might apply if this is not the case.

Now we know where that data is, how can we use it? In order to access the data one more important convention must be considered. When QuickBasic passes data to a routine it does not pass the value of the data item – it passes its address. Therefore, the value that is extracted from the stack at BP+6, BP+8 and so on must be used as a 2-byte address, at which the actual value will be found. Just what will be found at this address depends on the type of data item that was passed to the routine. Thus the second, third and fourth lines of a typical routine that has at least one (short) integer argument will be –

```
MOV BP,SP
MOV BX,[BP+6]
MOV AX,[BX]
```

or, in English, 'Copy the contents of the stack pointer into register BP. Then copy into register BX the value found at memory location BP+6. This value is the address of an integer variable, so copy into register AX the value found at memory location BX'. The use of register BX for the address is typical, while the use of register AX as the destination for the data item is arbitrary. If the routine has more than one argument then similar code will be used to access these additional items.

The final aspect of the routine's interface to the calling program is the return. This has previously been mentioned as RET *n*, where *n* is the unknown number. This magic number specifies how many bytes of data stored on the stack will be thrown away when the stack is cleaned up as the routine terminates. Because each argument passed from the calling program is a 2-byte address, *n* can be calculated as 2 times the number of arguments.

This form of the RET instruction does this reclaiming of stack space automatically. Note that this number does not include the return address, which is also on the stack – the argument to the return instruction is the number of *additional* bytes of data to remove from the stack.

Now we have an outline for the routine, we can fill in some of the interior. In order to do this we need a concrete example, so let's construct a simple one. We will build a SUB procedure called BITSET that sets a nominated bit within an integer to 1. A companion SUB procedure call BITCLR will set the nominated bit to 0. In line with Assembly language convention, the bits in

QuickBasic 4.5

MICROSOFT has released QuickBasic Version 4.5 with a number of enhancements aimed at first-time programmers. The new QB Advisor, an on-line reference system that uses 'hypertext technology', puts a wealth of information at the user's fingertips. Placing the cursor on a statement in a program, an error message, a menu choice – virtually anywhere on the screen – and pressing a mouse button, gives a complete explanation of the item.

The help facility also provides an ASCII chart and information on the limits of the programming environment. A new Instant Watch capability allows users to immediately find out the value of any variable or expression at any time.

Another new feature that helps the first-time programmer, the Easy Menus option, simplifies the user interface by reducing the number of choices in menus and dialog boxes. When QuickBasic is installed, it defaults to short, simplified menus, designed to let new programmers begin programming with a minimum set of commands. Once they have mastered the basics they can choose to view the full range of menu choices.

QuickBasic 4.5 is fully compatible with Microsoft's Basic Compiler Version 6.0. Minimum system requirements for are an IBM PC or compatible with 384 kilobytes memory, MS-DOS 2.1 or higher, and either two 360 Kbyte 5¼ inch drives or one 720 Kbyte 3½ inch drive.

Microsoft QuickBasic 4.5 is available in 3½ inch and 5¼ inch versions. Priced at $152 (including tax) it comes with a 30-day money-back guarantee. Upgrade pricing is $35 for registered users of Version 4.0 and $70 for registered uses of earlier versions.

Please check current pricing with your dealer.

an integer will be numbered from 0 to 15, low-order bit to high. The format will be BITSET(I%, N%) where I% is the number and N% is the bit position to be set. If N% is greater than 15, it will be truncated.

The procedure will operate like this: a single 1 bit will be loaded into the low-order position of the accumulator. Then the second argument (N%) will be used to count the number of times that this bit will be shifted left. This results in a number in the accumulator that is all zeroes, except for the bit that we want to set. This number can be ORed with the first argument (I%) to produce the result.

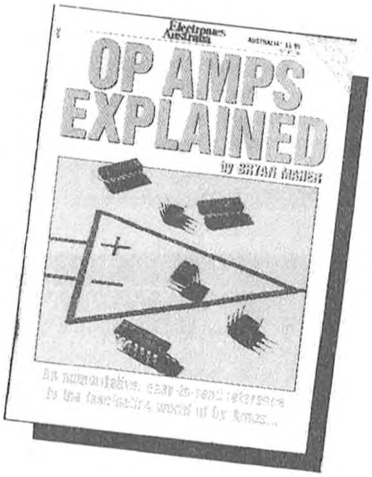
The complete routine is presented in Listing 1. Lines 1 to 4 are typical for a routine that accesses an integer as a argument – in this case it is the second (and last) argument and it has been loaded into register CX. Lines 5 and 6 put a 1 in the accumulator and shift it left the appropriate number of places. Lines 7 and 8 get the first argument into register CX, overwriting the previous value which is no longer needed. Line 9 does the logical OR, which has the effect of forcing the selected bit to 1. Note that the 'destination' register – the one that gets altered – is CX.

Now the result has to be returned. Because this is a SUB procedure, the result is returned to the calling program by placing

```
TITLE BITS.ASM - QuickBasic 4 Library Routine

.MODEL MEDIUM
DOSSEG
PUBLIC    BitSet
;***********************************************
;* FUNCTION BITSET (I%, N%)                      *
;* FORCE BIT NUMBER N% IN INTEGER I% TO 1.          *
;***********************************************

BitSet    PROC                      ;Procedure starts here.

          push   bp                 ;1  Save BP register.
          mov    bp,sp              ;2  Make BP into frame pointer.
          mov    bx,[bp+6]          ;3  Use frame pointer to get the
          mov    cx,[bx]            ;4   second parameter into CX.
          mov    ax,1               ;5  Load 1 into register AX
          shl    ax,cl              ;6   and shift it left N% places.
          mov    bx,[bp+8]          ;7  Get the first parameter
          mov    cx,[bx]            ;8   into CX.
          or     cx,ax              ;9  Force the bit on.
          mov    [bx],cx            ;10 Replace the parameter.
          pop    bp                 ;11 Retrieve BP and
          ret    4                  ;12  return past two arguments.

BitSet  ENDP                        ;Procedure ends here.
        END                         ;Module ends here
```

Listing 1. The complete listing for the sub procedure BITSET that sets a nominated bit within an integer to 1.

Continued on page 92.



Assembling QuickBasic - Part 2

Having covered QuickBasic routines in January, Jeff Richards continues with procedures for creating Assembly language function routines – they are very similar to those used to create Subs, but functions can be easier to use in QuickBasic programs.

BEFORE considering FUNCTION procedures and the way they differ from SUB procedures, a comment on the format of the arguments to the routine is in order. When calling such a procedure from a QuickBasic program it is usual to insert a variable as the argument: BITSET A,B. for example. However, any argument of the correct type can be used – in these examples this means anything that is an integer. Therefore the arguments can be variables, literals, constants or expressions. Thus, you could use BITSET A,1, BITSET 0,1 or BITSET 5,B+2.

While each of these forms is legal, they may not all be useful. If a procedure modifies an argument and returns it to the program, then a procedure call that uses a literal, a constant or an expression for the argument will appear to fail. What happens is that when a literal, constant or expression is used QuickBasic creates and initializes a temporary variable for that argument, and it is the address of this temporary that is passed to the procedure. The procedure will then alter this temporary by storing a result in it, but when it terminates and control is passed back to the calling program, the temporary is discarded, and the result cannot be retrieved.

This is obvious in the case of a literal or expression, but it's easy to pass a constant as an argument, and then spend a lot of time wondering why the procedure doesn't seem to alter the argument at all. This is one of several reasons why functions are often preferable to Subs.

The structure of a function is identical to that of a Sub, except for the way the result is returned. For a function, the result of the procedure is left in the AX register when the procedure returns. If it has been defined correctly, QuickBasic knows that the AX register contains the result, and it is presented to the calling program as the value of the function.

```
D:\>masm bits;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights
reserved.

  51560 + 271796 Bytes symbol space free

        0 Warning Errors
        0 Severe Errors
D:\>link /q bits
Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

Run File [BITS.QLB]: mylib.qlb
List File [NUL.MAP]:
Libraries [.LIB]: bqlb40.lib

D:\>
```

Figure 1. This is the actual display of the compile/link process to turn BITS.ASM into MYLIB.QLB.

Functions aren't prevented from returning results in their arguments, and this technique is often used for complex procedures that return multiple results. One result – an error code perhaps – is returned as the value of the function, while the others are returned in the arguments. However, this technique is rare in Basic – functions usually return a single result as the function value, and statements (which are equivalent to a Sub) don't usually modify their arguments (although Swap is an exception to this).

For an example of a function we will extend the BITSET and BITCLR examples from Part 1 (Feb '89) to include a test function, BITTST. This function will return 'true' if the nominated bit in the integer is set, otherwise it will return false. The full routine is presented in Listing 1.

```
TITLE BITS.ASM - QuickBasic Library Routine
DOSSEG
PUBLIC  BitSet, BitClr, BitTst

;**********************************************
;* FUNCTION BITSET (I%, N%)                   *
;* FORCE BIT NUMBER N% IN INTEGER I% TO 1.    *
;**********************************************
BitSet  PROC                    ;Procedure starts here.
        push    bp              ;1  Save BP register.
        mov     bp,sp           ;2  Make BP into frame pointer.
        mov     bx,[bp+6]       ;3  Use frame pointer to get the
        mov     cx,[bx]         ;4   second parameter into CX.
        mov     ax,1            ;5  Load 1 into register AX
        shl     ax,cl           ;6   and shift it left N% places.
        mov     bx,[bp+8]       ;7  Get the first parameter
        mov     cx,[bx]         ;8   into CX.
        or      cx,ax           ;9  Force the bit on.
        mov     [bx],cx         ;10 Replace the parameter.
        pop     bp              ;11 Retrieve BP and
        ret     4               ;12 return past two arguments.
BitSet  ENDP                    ;Procedure ends here.

;**********************************************
;* FUNCTION BITCLR (I%, N%)                   *
;* FORCE BIT NUMBER N% IN INTEGER I% TO 0.    *
;**********************************************
```

```
BitClr  PROC                    ;Procedure starts here.
        push    bp              ;1  Save BP register.
        mov     bp,sp           ;2  Make BP into frame pointer.
        mov     bx,[bp+6]       ;3  Use frame pointer to get the
        mov     cx,[bx]         ;4   second parameter into CX.
        mov     ax,1            ;5  Load 1 into register AX
        shl     ax,cl           ;6   and shift it left N% places.
        mov     bx,[bp+8]       ;7  Get the first parameter
        mov     cx,[bx]         ;8   into CX.
        not     ax              ;9  Invert the mask bits.
        and     cx,ax           ;10  and force the bit off.
        mov     [bx],cx         ;11 Replace the parameter.
        pop     bp              ;12 Retrieve BP and
        ret     4               ;13 return past two arguments.

BitClr  ENDP                    ;Procedure ends here.

;***********************************************
;* FUNCTION BITTST (I%, N%)                    *
;* RETURN TRUE IF BIT N% IN INTEGER I% IS SET, *
;   ELSE RETURN FALSE.                         *
;***********************************************

BitTst  PROC                    ;Procedure starts here.
        push    bp              ;1  Save BP register.
        mov     bp,sp           ;2  Make BP into frame pointer.
        mov     bx,[bp+6]       ;3  Use frame pointer to get the
        mov     cx,[bx]         ;4   second parameter into CX.
        mov     ax,1            ;5  Put a 1 in the low bit
        shl     ax,cl           ;6   and shift left N% places.
        mov     bx,[bp+8]       ;7  Get the first parameter
        mov     cx,[bx]         ;8   into CX.
        and     ax,cx           ;9  Test the bit and set flags.
        jz      TstEnd          ;10 Skip if zero flag set
        mov     ax,-1           ;11  else load -1 into result.
TstEnd: pop     bp              ;12 Retrieve BP and
        ret     4               ;13 return past two arguments.

BitTst  ENDP                    ;Procedure ends here.
        END                     ;Module ends here.
```

***Listing 1.** An example function – it extends the BITSET and BITCLR examples given in February to include a test function, BITTST. This function will return 'true' if the nominated bit in the integer is set, otherwise it will return false. This listing includes the Bit-Clear version of the bit-set program presented in February; it follows the same structure as the first program, but differs in the way the result is calculated. Since the 80x86 CPU does not support a NAND operation, it must be done in two steps. Firstly, the mask is inverted with the NOT instruction, then it is ANDed with the supplied number. The result is to clear the indicated bit, rather than setting it. If in doubt, try it with pencil and paper.*

As an exercise, try rewriting BITSET and BITCLR so that they are functions as well as Subs: that is, the argument is modified, but is also returned in AX. What advantages does this give for using the function/SUB in a program?

The first thing to note is that this function is almost identical to the BITSET procedure. The AX register is set up as a bit mask, with the nominated bit set and all others cleared. It is then ANDed with the integer argument, but this time the AX register is the destination. This means that the result of the AND will be a zero value in the AX register if the bit was not set, and an unchanged value if it was. More importantly, however, the Zero flag in the CPU flags register will be set if the result in AX is zero, and this flag can be used to control a conditional jump that will insert the True or False condition into AX, ready for the return.

Actually, only a False value has to be inserted. If the Zero flag is set, then the result in AX is zero and the bit was not set. A result of zero corresponds to False (as far as QuickBasic understands true and false) so no further action is needed. However, if the Zero flag is not set, and the result of the bit test is true, then we need to load a value of -1, which is True to QuickBasic, into AX.

```
DEFINT A-Z
DECLARE FUNCTION BITTST%(I%, N%)
WHILE  A <> -1
    INPUT "Number = ",A
    FOR I = 15 TO 0 STEP -1
        IF BITTST%(A, I) THEN PRINT"1";ELSE PRINT"0";
    NEXT I
    PRINT
WEND
```

***Listing 2.** A demonstration of the BITTST function – it displays the number entered in binary.*

Having tested the Zero flag and loaded AX accordingly, the routine is nearly finished. All that is needed is to retrieve BP from the stack and execute a RETurn with a parameter of 4, indicating that the function was called with 2 arguments.

Of course, using a function is different from using a Sub. Firstly, functions must be DECLAREd (this is optional, though recommended, for Subs). Secondly, functions cannot be used as stand-alone statements, but must be part of an expression. Thirdly, functions have a type, just as any other variable does. A program to demonstrate the BITTST function is presented in Listing 2 – this displays the number entered as binary. □

A note on style

THE PRESENTATION of a program can make a significant contribution to its clarity, and most programmers follow some sort of rule about style to help minimize the need for additional comments. The standard used here is –

Assembler directives – all upper case;

Labels and Variable names – capitalized, with embedded capitals to give a clue to their construction;

Mnemonics and registers – all lower case.

Names are kept short, and do not use any special characters. Some programmers prefer longer, more descriptive names, but often typing and spelling errors mean that long names cause more trouble than they are worth.

Assembling QuickBasic - Part 3

Once the basic requirements for constructing user-written assembly language routines for QuickBasic have been mastered, the number of different procedures and functions that can be implemented is limited only by imagination. Jeff Richards continues. . .

BEFORE CODING-UP a huge variety of useful routines, it is worth pausing a moment to consider whether or not the task really deserves a custom-written assembly language routine. For those procedures that are difficult or impossible to do from within QuickBasic, the decision is easy – an assembly language routine can often make the task simple. But most things the programmer needs to do can be done from within Basic – a special library routine may make them easier or quicker, but then it may not.

Although we can expect that an assembly language routine will be quicker than QuickBasic code, the optimisations that QuickBasic is able to perform mean that the distinction is not clear-cut. Because QuickBasic can intelligently examine the source code at the time it is compiled, extensive optimisations are possible. Statements that convert directly into machine language, such as AND and OR are usually implemented directly rather than as subroutine calls.

On the other hand, QuickBasic does not 'know' anything about our custom routine or how it works. Therefore, it cannot apply any optimisations at all. Each time the routine is called the arguments must be checked, temporaries created if necessary, arguments pushed onto the stack, and a call and simple routines are marginal as to whether or not they improve the speed of the final program. The longer and more complex a routine is, the more likely it is that a speed improvement will result.

Of course, there are reasons other than speed improvements for considering (or rejecting) assembly language routines. A simple routine call that replaces several lines of mysterious code can make programs more compact and comprehensible. But it can also make programs quite unintelligible to someone who doesn't have a reference to the routines that have been used! This month's routine might be on the marginal side with respect to speed, but it illustrates an important facility that is available with assembly language routines that may be significant for certain types of procedures –the ability to pass a variable number of arguments.

To illustrate the concept, I will use a simple FUNCTION routine called MAX that finds the largest of a number of integer argu-

```
TITLE MAX.ASM - QuickBasic Library Routine

DOSSEG

PUBLIC Max
;*********************************************
;* FUNCTION MAX(I1%, I2% . . . . N%)            *
;* RETURN THE VALUE OF THE LARGEST INTEGER IN   *
;    THE ARGUMENT LIST In%.  N% MUST BE THE     *
;    NUMBER OF ITEMS IN THE LIST.               *
;*********************************************
Max     PROC                    ;Procedure starts here.
        push    bp          ;1  Save BP register.
        mov     bp,sp        ;2  Make BP into frame pointer.
        mov     bx,[bp+6]    ;3  Use frame pointer to get the
        mov     cx,[bx]      ;4   argument count into CX.
        push    cx           ;5  Save it for Ron.
        mov     ax,8000h     ;6  Clear AX to MIN.
        mov     di,8         ;7  Set DI to last list item.
XLoop:  mov     bx,[bp+di]   ;8  Get next list item
        mov     dx,[bx]      ;9   into DX.
        add     di,2         ;10 Increment DI pointer by 2.
        cmp     ax,dx        ;11 Is AX (= than DX?
        jnle    XSkip        ;12 No - skip.
        mov     ax,dx        ;13 Yes - load DX into AX.
XSkip:  loop    XLoop        ;14 Repeat until CX = 0.
        pop     bx           ;15 Retrieve argument count.
        inc     bx           ;16 Add 1 for the count itself.
        shl     bx,1         ;17 Make it a byte count.
        pop     bp           ;18 Retrieve BP
        pop     cx           ;19 Pop return segment and
        pop     dx           ;20  address.
        add     sp,bx        ;21 Adjust the stack pointer.
        push    dx           ;22 Replace the return segment
        push    cx           ;23  and address.
        ret                  ;24 Do a simple far return.

Max     ENDPT                ;Procedure ends here.
END                          ;Module ends here.
```

This simple FUNCTION routine illustrates an important facility that is available with assembly language routines may be significant for certain types of procedures – the ability to pass a variable number of arguments. The routine, called MAX, finds the largest of a number of integer arguments. So the routine knows how many arguments to examine, the argument count will be passed as the last parameter. The routine starts in the conventional manner, and loads the last parameter – the argument count – into CX. It saves this on the stack for use later on (line 5). Register AX is initialised to the smallest integer value (line 6) and DI is set to 8 to point to the next argument – the last list item (line 7). Each argument is accessed by using BP+DI to point to its address (line 8), and loading it into DX (line 9). Each time through the loop AX is replaced with the argument only if it is larger than the current value (lines 11 to 13). DI is incremented by 2 to point to the next list item (line 10), and the LOOP opcode automatically decrements CX until it reaches zero (line 14).

ments. So that the routine knows how many arguments to examine, the argument count will be passed as the last parameter. Conveniently, this is the the only one the routine can be certain of finding, given that it doesn't know how many arguments there are. Listing 1 shows the complete procedure.

Notice in the listing, how DI is used to point to each successive argument in the parameter list. DI actually stands for Destination Index, but when used in this manner it can be thought of as a Displacement. There are other registers that could have been used, but this is a typical task for DI. It is at the finish of the routine that the complexity arises. The problem comes from the fact that the RET instruction needs a parameter that is used to clear the arguments off the stack. At the time of writing the routine we don't know how many arguments there are, so the RET instruction cannot be coded correctly.

Stack for storage?

THE IDEA of using the stack within the routine for temporary storage of data may seem a little strange – after all, we are trying to access items of data that have been pushed onto the stack by the calling routine. Well, that's true, except the items that have been pushed onto the stack will not be accessed by popping them off the stack. The value of the stack pointer when the routine commenced has been copied into BP, and we access the items by referring to locations BP+6, BP+8 and so on. Anything that may be done with the stack pointer as a result of PUSHing additional items will have no effect on our ability to find the data we need using offsets from BP.

There are a number of ways of dealing with this, but the one presented here is probably the simplest. The program knows how many items were in the list – this was loaded into CX and saved on the stack. It can now be popped off the stack (line 15) and by adding 1 (line 16) we have the total number of arguments. Multiplying by 2 (line 17) gives us the number of bytes to be cleared off the stack when the routine returns. This value can be used to adjust the stack pointer, but we can't simply add the byte counter to SP. This is because the return address we need to get back to the calling routine is at the top of the stack. So, before doing anything we must retrieve from the stack everything we need – register BP (line 18) and the return address – segment and offset (lines 19 and 20). Now that there is nothing useful on the stack we can adjust it for the function arguments (line 21). The easiest way to use the return address to get back to the calling program is simply to push it back onto the stack and execute a RET with no argument (lines 22, 23 and 24).

An alternative procedure could involve actually writing the calculated argument into the RET instruction. This is called 'self-modifying code', and can be very dangerous for machines that utilise a pre-fetch instruction queue to speed up processing. Although there are ways around this difficulty, it is simpler to manually clean up the stack as in this example.

Assembling this routine and building it into a library is done in the same way as the others that have been presented. However, using it is slightly different. Firstly, when it is declared, the parameter-type checking QuickBasic performs must be disabled. This is done by leaving the parameter details out of the declaration. The required line of code is –

```
DECLARE FUNCTION MAX%
```

Then, when the function is used, the number of list items must be included as the last argument. Failure to do this correctly will cause the program to go off into never-never land, from which only the Big Red Button will rescue it. A typical usage would look something like this

```
IF A <> MAX%(A, B, C, D, 4) THEN . . .
```

Like many assembly language routines, the procedures involved in correctly starting up the routine, and then exiting in a correct and tidy manner, are often more complex than the procedures that actually do the useful work. However, once these procedures are worked out for a certain class of routine, them implementing different versions for different problems is easy. Converting the MAX function to a MIN would be trivial, and implementing something like an AVERAGE function would be only slightly more difficult. □

Assembling QuickBasic - Part 4

Although the assembly language routines for QuickBasic presented so far have all used integer variables, there is no reason that these routines should not make use of other variable types, or use arrays instead of simple variables. String variables will be considered later, but this month Jeff Richards discusses using arrays of integers.

ARRAYS ARE ONE area in which Revision 4 of QuickBasic differs significantly from previous revisions. The major difference as far as assembly routines are concerned is that the address of an array element must be specified as a full segment/offset address (4 bytes) to ensure compatibility with both static and dynamic arrays, and environment and stand-alone programs.

This requirement should only be a problem when attempting to pass whole arrays to a procedure. If single elements are passed they can be forced into temporaries, which are treated as simple variables. There are two ways of forcing array elements into temporaries. One is to encloses the variable name in parentheses, such as (A(1)) or (N(B * 2)). Another is to make it an expression, for instance by adding zero – A(1) + 0. Of course, any problems in using arrays can be avoided simply by assigning the value of the array element to a simple variable and using the simple variable as the argument to the routine.

The biggest problem in using array elements arises from SUB procedures that return results in their arguments. Forcing array elements into temporaries for these routines will not work, as the returned result which is loaded into the temporary is lost when the temporary is discarded as the routine returns. The only solution for this situation is to transfer the array element into a simple variable, use the simple variable in the routine, and copy the result back into the array.

But when you want to write a function or subprogram that refers to a number of sequential elements from an array, the best solution is to pass to the procedure the full, double-word, array address. This address is found using the VARPTR and VARSEG functions on the first array element. VARPTR provides the offset address of the array element, while VARSEG provides the segment address.

Listing 1 is the MAX function rewritten to return the maximum value of a sequence of array elements. The arguments to the function are the address of the first array element (offset and segment) and the number of elements to consider. Although the process of calculating the addresses of the data is more complex, the whole difficulty of cleaning up the stack on exit is eliminated because of the fixed number of arguments.

Lines 4 and 5 get the number of list elements into CX, as in the previous example. Then the first and second arguments are loaded into the ES:DI register pair. ES is a segment register, while register DI, when used as an address, is assumed to refer to an offset in the ES segment. Therefore, loading ES and DI with the segment and offset address of the first array element neatly solves the problem of passing a full double-word address for arrays.

The routine executes the actual comparison in a similar manner to the MAX routine presented in Part 3. Each array element is accessed using ES:DI at the start of the loop (line 11) before being compared to AX. Register DI is incremented by 2 each time so that ES:DI points to successive elements of the array. When the loop terminates, the largest value in the array is left in AX, and returned to the calling program.

Note that register DI is saved on the stack at the start of the program, and retrieved at the end. Although this is not strictly necessary, it is recommended. Also note that it is saved on the stack after loading BP from SP. This enables us to maintain consistent offsets from BP (6, 8, 10 and so on) for the arguments.

Such a function would be declared with a line like –

```
DECLARE FUNCTION AMAX% (I%, J%, K%)
```

and used in a line such as –

```
A% = AMAX% (VARPTR(A%(0)), VARSEG(A%(0)), 10)
```

Note that the actual value of VARSEG(A%(0)) and VARPTR(A%(0)) can change while the program is running, so they must be assigned immediately before the function is called. The easiest way to ensure this happens is to embed the functions within the argument list, as in the example above.

Gaps in Parts?

MISSED PARTS of any of our YC Tutorials? Well, you've got a second chance. We'll forward photocopies of one to five parts for $4 including postage and handling, and six and over parts for $8 (overseas postage additional). Please identify the Tutorial by its complete name, the issue it appeared in and the Part number (if applicable). All of our recent Tutorials are listed in YC's Annual Index, in this issue: refer to the Services page on how to obtain Back Issues.

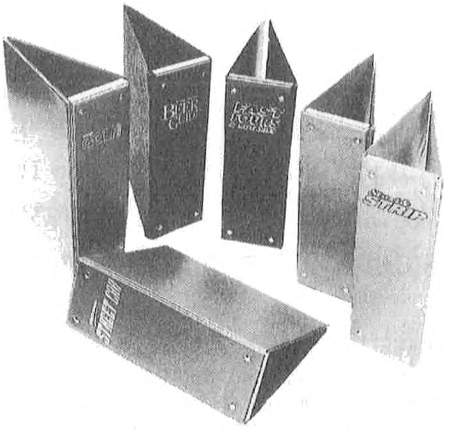
```
TITLE AMAX.ASM - QuickBasic Library Routine

.MODEL MEDIUM
DOSSEG
.CODE
PUBLIC AMax
;************************************************
; FUNCTION AMAX(Ptr%, Seg%, N%)                 *
; RETURN THE VALUE OF THE LARGEST OF N% INTEGERS*
;    IN THE ARRAY STARTING AT OFFSET P% IN      *
;    SEGMENT S%.                                *
;************************************************

AMax    PROC                      ;Procedure starts here.

        push    bp                ;1  Save BP register.
        mov     bp,sp             ;2  Make BP into frame pointer.
        push    di                ;3  Save DI Register
        mov     bx,[bp+6]         ;4  Use frame pointer to get the
        mov     cx,[bx]           ;5   list count into CX.
        mov     bx,[bp+8]         ;6  Get Segment of array
        mov     es,[bx]           ;7   into ES.
        mov     bx,[bp+10]        ;8  Get Offset of array
        mov     di,[bx]           ;9   into DI
        mov     ax,8000h          ;10 Clear AX to MIN.
AXLoop: mov     dx,es:[di]        ;11 Get next list item into DX
        add     di,2              ;12 Increment DI pointer by 2.
        cmp     ax,dx             ;13 Is AX <= than DX?
        jnle    XSkip             ;14 No - skip.
        mov     ax,dx             ;15 Yes - load DX into AX.
XSkip:  loop    AXLoop            ;16 Repeat until CX = 0.
        pop     di                ;17 Retrieve DI
        pop     bp                ;18 Retrieve BP
        ret     6                 ;19 Return past 3 arguments.
AMax    ENDP                      ;Procedure ends here.
        END                       ;Module ends here.
```

***Listing 1.** The MAX function from Part 3 (May 1989) rewritten to return the maximum value of a sequence of array elements. The arguments to the function are the address of the first array element (offset and segment) and the number of elements to consider.*

Of course, it is not necessary that the address used is that of the first array element – it is quite possible to find the maximum value of a subset of the array by passing the address of the first element of the subset, and the number of elements in the subset. For instance, a line such as –

```
A% = AMAX% (VARPTR(A%(5)), VARSEG(A%(5)), 5)
```

will find the maximum value in the range of items A(5) through to A(9). The Microsoft documentation doesn't say so, but it can be assumed that the segment address of every element in the array will be the same, so it wouldn't matter whether VARSEG(A%(0)) or VARSEG(A%(5)) was used in the above example. □



ASSEMBLING QUICKBASIC – Part 5

So far in the discussion of assembly language routines for QuickBasic, only integers have been considered. Of the other possible variable types, strings are most likely to be useful. Jeff Richards continues. . .

ALTHOUGH QUICKBASIC provides an extensive range of functions for string handling, each one extracts a considerable overhead in processing time. New string storage areas must be allocated, strings copied from source to destination, and unused areas reclaimed. Specialised assembly language routines for string handling can make a significant improvement to the performance of QuickBasic.

There are limits on the type of string operations that can be performed in assembly language routines. As a general rule, any operation that alters the length of the string cannot be implemented in an assembly language routine. They are therefore most useful for operations that scan strings for characters or patterns, or perform string character conversions *in situ*.

The QuickBasic example in Listing I will scan a string and check that all characters in the string fall into a specified range of ASCII values. The routine could be used to scan user input to check that, for instance, the string consisted of all numeric characters.

The equivalent assembly language procedure is presented in Listing 2. The difference in dealing with strings compared with integers – apart from the actual operations performed – arises from the way the string is accessed. While obtaining the value of an integer variable is a two-stage process, accessing the characters of a string variable involves three separate steps.

The first step is to obtain the value of the argument passed from the calling program. Remember that QuickBasic passes all arguments as two-byte addresses, and this includes strings. In the example presented here, the string is the first of three arguments; it was therefore pushed onto the stack first, so it will be found three levels (six bytes) up. The stack offset is four bytes, so the code 'mov bx,[bp+10]' will load the value of the string argument into register BX.

```
FUNCTION CHECK% (I$, N%, M%) STATIC
     F% = -1
     FOR I%=1 TO LEN(I$)
          X% = ASC(MID$(I$, I%, 1)
          IF X% ( N% OR X% ) M% THEN F% = 0
     NEXT I%
     CHECK% = F%
END SUB
```

***Listing 1.** This QuickBasic program will scan a string and check that all characters in the string fall into a specified range of ASCII values. Although this function consists of just a few lines, it includes a large number of calls to built-in functions, and is relatively slow.*

This value is an address (remember, QuickBasic passes all. . . and so on) so the value we are after is actually [bx], or 'the contents of the memory location referred to by the value in register BX.' But this is not the string! It is instead a data structure called

```
ALoop: cmp   byte ptr [bx],al ;11 Check against lower limit
       jb    Errx             ;12  and take error exit if fail
       cmp   byte ptr [bx],ah ;13 Check against upper limit
       ja    Errx             ;14  and take error exit if fail
       inc   bx               ;15 Bump the pointer
       loop  ALoop            ;16  and repeat until end-of-string
AExit: mov   ax,-1            ;17 Return TRUE
       pop   bp               ;18
       ret   6                ;19
Errx:  xor   ax,ax            ;20 Return FALSE
       pop   bp               ;21
       ret   6                ;22
Check  ENDP
DOSSEG
PUBLIC Check
;********************************************
;* FUNCTION CHECK% (I$, N%, M%)             *
;* RETURN TRUE IF NO CHARACTERS OF I$ ARE   *
;*  OUTSIDE THE (ASCII) RANGE N% TO M%.     *
;********************************************
Check  PROC
       push    bp             ;1  Save Stack Pointer
       mov     bp,sp          ;2  Make BP the frame Pointer
       mov     bx,[bp+10]     ;3  Get address of string descriptor
       mov     cx,[bx+0]      ;4  Get string length into CX
       jcxz    AExit          ;5  Exit if null string
       mov     bx,[bx+2]      ;6  Get address of 1st character
       mov     dx,[bp+8]      ;7  Get low parameter
       mov     al,[dx]        ;8   into AL
       mov     dx,[bp+6]      ;9  Get high parameter
       mov     ah,[dx]        ;10  into AH
```

***Listing 2.** The assembly language equivalent of Listing 1.*

a string descriptor. It is a very simple data structure, consisting of just two words. The first is the current length of the string. The second is the address at which the first character of the string will be found.

So we have BX pointing to a string descriptor that consists of the string length and the address of the first character. The code 'mov cx,[bx]' loads the length of the string into CX. (To code this as 'mov cx,[bx+0]' is pedantic, but perhaps makes it clearer that we are actually accessing the first word of the string descriptor.) Line 5 tests this value, and if the string is currently null the routine branches straight to the end. Note that a null string causes the function to return 'True'.

The choice of the CX register to hold the string length is not arbitrary – it will be used as the loop counter when the string is examined character by character. If the string has characters to process, then we can continue on to the third stage of accessing the characters in the string. BX still points to the string descriptor. Therefore the first character of the string is at the address stored at location BX+2 – the second word of the string descriptor. In assembler notation this is referred to as [BX+2], or 'the contents of the memory location referred to by the value stored two bytes beyond where BX is currently pointing.' Line 6 loads this address into BX, overwriting the address of the string descriptor. Loading a register with the contents of an address calculated using the current contents of that same register may look strange, but it is perfectly legal.

The above description involves an addressing procedure that could be called 'Double indirect.' A diagram makes it much clearer – if 'TEXT' was passed as the first of three parameters, then the memory layout would look like this:

```
Increasing       Stack       String                String
Memory                       Descriptor            Storage
Addresses                                          Byte 3 "T"
           SP+10 Arg 1 --                          Byte 2 "X"
  ¦        SP+8  Arg 2 ¦                           Byte 1 "E"
  ¦        SP+6  Arg 3 ¦     Word 1 (uf1970) Address --) Byte 0 "T"
  ¦        SP+4  Word  ¦--)  Word 0 (uf1970) Length (4)
  ¦        SP+2  Word
  ¦        SP -) Word
```

AL and AH are loaded with the other two arguments (using DX as the index register – BX is otherwise engaged) and the string is checked character by character. Note that the characters in the string are not actually loaded into a register – the compare instructions in lines 11 and 13 compare AL and AH to the memory location pointed to by BX.

Because the string length was loaded into CX, the LOOP instruction can be used to automatically examine the correct number of characters. However, register BX has to be explicitly incremented each time through the loop in order to address each character in turn. If the loop gets to its end without an error, then a value of TRUE (-1) is loaded into AX and the routine terminates. If a character was found that is outside the legal range, then a jump to Errx loads a result of FALSE and terminates the routine. □

Continued from page 84.

the result into the address of one of the arguments – in this case the first argument. Conveniently, this address is still held in the BX register, so Line 10 simply copies the result from the CX register into memory at the location pointed to by the BX register.

Finally, to finish the procedure, BP is popped off the stack and a return instruction executed. Because there were two arguments passed to the routine, the argument to the RET instruction is 4.

The parameters passed to the routine are thrown away when the routine terminates, but remember that these were the addresses of the variables involved. Therefore, the updated value – the result of the routine – that was loaded into the first argument is still there and is accessible to the calling program.

To complete the process, the routine needs to be built into a library. With the routine in a file called BITS.ASM and with MASM.EXE in the current directory or search path, just type MASM BITS; – no special options are required. Note that MASM will choke on the file if it is not plain ASCII. For instance, WordStar in N mode is suitable, but D mode is not. If in doubt, use EDLIN or a similar text editor.

To create the quick library, make sure that LINK.EXE is in your current directory or searchpath, and that BQLB40.LIB is also in the current directory. Then type LINK /Q BITS.OBJ, MYLIB.QLB, , BQLB40.-LIB;. This will create a Quick library called MYLIB.QLB that contains only one routine – BITSET. To see the action a little more clearly, it is possible to type LINK /Q BITS-.OBJ and then answer the questions one at a time. The Run file is MYLIB.QLB, the List file is NUL.MAP (you can accept the default) and the Library is BQLB40.LIB.

To use the new library, start QuickBasic with the command QB /LMYLIB.QLB. The program in Listing 2 shows how to declare the SUB procedure, and demonstrates how to use it. Try it with values such as 0,0 or 0,2.

Although this example doesn't do anything that can't be done from within QuickBasic, the procedures outlined here will work for much more complex examples. However, the real convenience of user-written Assembly language subroutines comes when they are written as functions. Functions differ from SUBs in that they return a result directly to the calling program. The additional steps required for functions are not complex – in some respects they are easier than SUBs. □

A note on style

THE PRESENTATION of a program can make a significant contribution to its clarity, and most programmers follow some sort of rule about style to help minimize the need for additional comments. The standard used here is –

Assembler directives – all upper case;

Labels and Variable names – capitalized, with embedded capitals to give a clue to their construction;

Mnemonics and registers – all lower case

Names are kept short, and do not use any special characters. Some programmers prefer longer, more descriptive names, but often typing and spelling errors mean that long names cause more trouble than they are worth.

ASSEMBLING QUICKBASIC - Part 6

Jeff Richard's discussion of assembly-language functions for QuickBasic has so far assumed they return only integer results. However, the work involved in constructing functions that return strings is only slightly more complex, and it certainly increases the range of useful assembly-language functions that can be built.

STRING HANDLING procedures are fast and effective in QuickBasic. However, this is achieved through complex procedures involving the allocation of string storage areas and the management of string free space. Virtually any string operation involves the copying of the string to a temporary region, the operation on the string, and the reallocation of the changed string to a new region. Each of these procedures may involve reclamation of unused string space, and an occasional restructuring of the whole of the string storage area may be required.

An assembly-language routine does not have access to these string management facilities, so the operations that can be performed on strings are limited. However, returning a string result in a function is certainly possible. The example presented here will convert a month number (1 to 12) into a month name.

The possible string results will be stored as constant data in the routine. In order to make the management of the strings as simple as possible, two rules about the storage of this constant data will be applied. Firstly, each string will take up the same number of bytes. This will make the process of discovering the start of the required string a simple procedure involving counting from the beginning of the storage region. Secondly, each string will be preceded with a single byte indicating its actual length. An alternative process for finding the actual length would be to include a special terminator in each string, but the length-byte approach is simple to construct and to use.

```
TITLE   MONTH   QuickBasic 4 Library Routines
DOSSEG


;*********************************************************
;* DECLARE FUNCTION MONTH$ (N%)                           *
;*  Return the name of month number N%.                   *
;*********************************************************

; 12 entries of 10 bytes each as (Length)STRING

M_Table DB      7,"January  "
        DB      8,"February "
        DB      5,"March    "
        DB      5,"April    "
        DB      3,"May      "
        DB      4,"June     "
        DB      4,"July     "
        DB      6,"August   "
        DB      9,"September"
        DB      7,"October  "
        DB      8,"November "
        DB      8,"December "

M_Strd  LABEL   WORD    ; String Descriptor structure.
M_StrL  DW      0       ; Default is empty string
M_StrO  DW      ?       ;  at an unknown address
```

```
PUBLIC Month
Month   PROC
        push bp                 ;1
        mov  bp,sp              ;2
        mov  byte ptr M_StrL,0  ;3 Force string length to zero
        mov  bx,[bp+6]          ;4 n = Month #
        mov  ax,[bx]            ;5
        dec  ax                 ;6 (range 0 - 11)
        cmp  ax,12              ;7 if n ) 11, then
        jnb  M_Exit             ;8  return null string.
        mov  bx,Offset M_Table  ;9  BX = Table Base
        shl  ax,1               ;10 AX = 2n
        add  bx,ax              ;11 BX = Base + 2n
        shl  ax,1               ;12 AX = n x 4
        shl  ax,1               ;13 AX = n x 8
        add  bx,ax              ;14 BX = Base + 2n + 8n
        mov  ax,[bx]            ;15 Load the length byte
        mov  byte ptr M_StrL,al ;16  into the descriptor.
        inc  bx                 ;17 BX points to the first
        mov  M_StrO,bx          ;18  character so load it also.
M_Exit:                         ;19
        mov  ax,offset M_StrD   ;20 Load the address of the
        pop  bp                 ;21  String Descriptor, and
        ret  2                  ;22  Return.
Month   ENDP
```

Listing 1. *An assembly-language routine does not have access to string management facilities, so the operations that can be performed on strings are limited. However, returning a string result in a function is certainly possible. This example will convert a month number (1 to 12) into a month name. A similar procedure could be used to convert a day number (0 to 6) into a day name.*

The table of month names is labeled 'M_Table' and is allocated within the region declared with the '.DATA' directive. Since the longest month name is 9 characters, each string will consist of the length byte and 9 bytes of the string. The actual month name will be left-justified within the 9 bytes. A single length byte followed by a 9-byte string gives a very convenient 10-byte length for each month name entry.

Immediately following the string data is the string descriptor data structure. This data structure is the key to being able to create functions that return strings. It is not a complex structure, consisting of just two words. The first word is labeled 'M_StrL' and will contain the length of the result string. The second word is labeled 'M_StrO' and will contain the offset address within the data segment of the first character of the string.

The function will return the string to QuickBasic by returning (in the AX register) the address of the string descriptor. Therefore, the start of the string descriptor data structure has been labeled with 'M_Strd' to make reference to it as simple as possible. Assigning a label like this does not allocate storage – it simply makes it easy to refer to the location using the label. The DW and DB directives assign the actual storage space.

Note that the string descriptor table is initialised with a string length of 0. If the string is empty, then the address of the first character is meaningless, and this is indicated by initialising the string address to '?'. Initialising the string descriptor in this way means that if the routine returns to QuickBasic without making any adjustment to the descriptor table, QuickBasic will see the result as a null string.

The process of calculating the month name and returning the string result therefore boils down to inserting the appropriate string length and pointer into the string descriptor table, loading the address of the table into register AX, and returning to QuickBasic.

To do this, the month number is accessed from the argument list, decremented by 1 to get it into the range of 0 to 11, and then multiplied by 10 so that it can be used as a pointer to the correct month name. This value is added to the address of the first month name, so the result is the address of the length byte of the required month. This length byte is loaded into the first word of the string descriptor. The address is incremented by one so that it points to the first character of the string and this address is then stored into the second word of the string descriptor.

The routine to multiply by 10 could be replaced with a MUL instruction, but this is extremely slow, especially on 8086/8088 machines. If the argument passed to the routine was outside the range 1 to 12, then this is detected at line 7 and the routine does not adjust the string descriptor. Finally, the address of the string descriptor is loaded into the AX register (line 20) and the routine terminates.

Functions that return strings can be used like any string variable. It is always wise to declare functions, so the program would start with a statement such as –

```
DECLARE FUNCTION MONTH$ (I%)
```

and the function would be used in code such as –

```
FOR I% = 1 TO 12
 _OCATE I%+2, 10
 PRINT MONTH$(I%)
NEXT I%
```

Since a string function does not use or allocate storage in the QuickBasic string storage area, some restrictions apply to the use of such functions. For instance, the statement –

```
MID$ (MONTH$(N%), I, 1) = "*"
```

would be rejected by the compiler. Such a statement is quite meaningless, as there is no way the program can subsequently access the resultant string. To achieve the required result, use a sequence such as –

```
A$ = MONTH$(N%)
MID$ (A$, I, 1) = "*"
```

Multiplying by ten

THE ROUTINE multiplies by 10 in a series of shifts and adds. This is a very old technique that can be used when the multiplier is fixed, and has factors that are a power of 2. In this case, n*10 is regarded as (n*2) + (n*8). To multiply by two in binary, simply shift left once. To multiply by 8, shift left three times. Therefore, multiplication by 10 can be described as (n shl 1) + (n shl 3) or (n shl 1) + ((n shl 1) shl 2). This is the formula in this routine. □

ASSEMBLING QUICKBASIC - Part 7

Jeff Richards describes accessing Dos and BIOS services in QuickBasic.

ASSEMBLER ROUTINES that access Dos services are among the most useful that can be written. The types of facilities provided by Dos are extensive, and considerable programming work can be saved by building special-purpose routines that take advantage of these services.

Many of the services available from Dos are already provided for in QuickBasic. For instance, all file management routines use Dos services, as do the printer commands. There are some Dos services, such as setting the current disk or directory, that do not have a direct counterpart in QuickBasic, and these are obvious candidates for assembler routines. But there are also Dos services that are provided for in QuickBasic, but can be more conveniently done through a special library routine. The first routine this month is an example of this – it will interrogate Dos to find out if a file already exists.

The usual procedure for finding if a file exists, is to attempt to open it for input. If the open succeeds then the file exists and it can be closed and re-opened in the correct mode. If the open fails the file does not exist, the program can take the correct action. This complex (and slow) procedure can be replaced with a library routine that consists of a single call that returns a value of true or false.

The routine FEXIST is constructed as a function that takes a string argument and returns an integer result. The result will be true (-1) if the file exists and can be opened, or false (0).

The routine is a simple one that essentially follows the QuickBasic procedure of attempting to open the file and returning a false result if the open fails. There is, however, one small complication. QuickBasic handles strings by keeping note of the current string length. Dos does not require information about the length of the string, but instead requires that the last character of the string is followed by a null character – binary 0. Library routines cannot add characters to the end of a QuickBasic string, so the filename passed to the library routine must first be copied into a local storage area before the null character is appended, and the filename passed to Dos for evaluation. This string checking and copying takes as much code as the actual Dos calls do.

After the initialisation steps, the string length byte is accessed from the string descriptor table in line 6. If the string is not too long, then the full string address is loaded into register DS:SI (line 10). The address of the buffer is then loaded into registers ES:DI (lines 11 to 14). Because the length of the string is already in CX, a REP MOVSB instruction can be used to copy the string. This instruction copies CX characters from DS:SI to ES:DI, incrementing SI and DI as it goes. Register DI will end up pointing one byte beyond the end of the string, so appending a null to the string simply takes one more instruction (line 16). Now we are ready to call Dos.

Calling Dos

CALLING DOS IS a simple process, but an accurate reference to Dos calls is required. The Microsoft *MS-Dos Programmer's Reference Manual* is the complete source for Dos call information, but there are many other sources, including some public domain files. The procedure is to load values into registers and then execute an interrupt. The interrupt for almost all Dos calls is 21h. The function to be executed is loaded into the AH register, and different services may require data to be loaded into other registers.

The example procedure will use function number 3Dh to open a file. The mode value, which is loaded into register AL, is zero, which means to attempt to open the file for input. The address of the file name must be in register pair DS:DX, which, conveniently, it already is. If the call succeeds then a file handle is returned in AX, and the file can be closed. The close function is 3Eh, with the file handle in register BX. Whether or not the open attempt is successful is indicated by the carry bit in the flags register. This is tested in line 21 with the JC command – Jump if Carry is Set.

Note line 20. To load a value of zero into a register, the XOR command is often used. This may be faster and smaller than MOV, but it has the effect of clearing the carry flag. At line 20 we are loading register AX in preparation for testing the result of the Dos call. An XOR would clear the carry flag and prevent us from detecting that the Dos call had failed. Loading zero into AX before the test makes the program a little simpler, but it also lays a trap for those who think they can make the program smaller and faster by using an XOR instead of a MOV.

At line 25, the AX register contains either a zero if the string was too long or the file was not found, or -1 if the open (and close) was successful. Returning to QuickBasic simply involves retrieving the saved registers and executing a RET with a value of 2. The function can be used as a simple expression that is TRUE if the file exists, for example –

```
IF FExist% ('DATA.FIL')
THEN GOTO OpenFile ELSE.(th).(th).
```

Note, however, that the function will return false if the file exists, but there are no spare file handles available for Dos to allocate. In this case, the QuickBasic OPEN would also fail. Therefore, the FExist% function should be used either before any files are opened, or immediately before the required file would have to be opened. The file name can include a disk and directory specifier, but the routine as presented places a limit of 63 characters on the length of the filename string.

Accessing BIOS Services

WHILE DOS SERVICES provides a wide range of facilities for file management tasks, BIOS services can be considered as operating at a much lower level. The BIOS services operate much closer to the level of the hardware – they are one step above direct manipulation of the memory and I/O ports. There are a large number of BIOS services available, and the main advantage of accessing them through assembly-language library routines is speed.

Many BIOS functions parallel the services provided by Dos. For instance, characters can be written to the display by calling BIOS interrupt 10h or Dos function 2. However, the facilities offered by the two alternatives will differ considerably. Generally, the BIOS will offer more facilities, but usually at the cost of added complexity. The original IBM technical reference manuals are the best references to the BIOS services available, but there are many other references.

> *The main advantage of accessing [BIOS services] through assembly-language library routines is speed.*

The work involved in making use of BIOS services can range from the trivial to the complex. For instance, a routine to print the screen, exactly as if the user had hit the PrtSc key, takes four instructions, two of which could be labeled 'housekeeping'! Information maintained by the BIOS in its data area is accessed simply be reading the location. On the other hand, a complex screen updating routine might require a series of BIOS calls to determine the monitor type, find the active display page, manage the cursor, set display defaults, update the screen, and restore the cursor.

Two examples of using the BIOS will be presented here. Both refer to the keyboard – one uses BIOS calls and the other refers to data in the BIOS data area. They both perform tasks that can be done from within QuickBasic, though not as efficiently, and in both cases the alternative QuickBasic code will be shown.

```
TITLE     KeyBoard     QuickBASIC 4 Library Routine
DOSSEG
;********************************************************
;* DECLARE SUB ClrKbd ()                                *
;* Soak up any pending keystrokes from the keyboard.    *
;********************************************************
PUBLIC ClrKbd
ClrKbd PROC
       push     bp
Again: mov      ah,1         ;Is there a character pending?
       int      16h
       jz       Finis        ;No - exit
       mov      ah,0         ;Else get the character
       int      16h
       jmp      SHORT Again ;And repeat
Finis: pop      bp
       ret
ClrKbd ENDP

;********************************************************
;* DECLARE FUNCTION KBDSTAT$ (MASK%)                    *
;*     Return Keyboard status word ANDed with MASK%.    *
;********************************************************
PUBLIC KbdStat
KbdStat PROC
      push     bp          ;Save BP.
      mov      bp,sp       ;Use BP as frame pointer.
      mov      ax,40h      ;Set ES to BIOS data
      mov      es,ax       ;  segment address.
      mov      ax,WORD PTR es:17h ;Get status byte.
      mov      bx,[bp+6] ;Get parameter address.
      and      ax,[bx]     ;AND with status byte.
      pop      bp          ;Retrieve BP
      ret      2           ; and return.
KbdStat ENDP
```

Listing 1. *The routine is a simple one that essentially follows the QuickBasic procedure of attempting to open the file and returning a false result if the open fails. There is, however, one small complication. QuickBasic handles strings by keeping note of the current string length. Dos does not require information about the length of the string, but instead requires that the last character of the string is followed by a null character – binary 0.*

The first routine is designed to clear the keyboard of all pending keystrokes. This is usually required when an error or unusual processing condition has occurred, and all processing should be suspended until a response is obtained from the operator. Clearing the keyboard helps ensure that the important prompt will not be missed as the operator types ahead of the program.

The QuickBasic code to achieve this is –

```
WHILE INKEY$ (cf83);(cf94)(uf1975)
          " " : WEND
```

The library routine is simply the statement CLRKBD. Like the FEXIST% function in listing 2, the library routine will follow the logic of the QuickBasic version. The BIOS will be interrogated to discover if there is a keystroke ready. If there is, it will be collected and discarded. This will continue until the BIOS says that there are no more keystrokes, and the routine will terminate. The routine takes no arguments and returns no results.

Bit	Key/Toggle	State
0	Right Shift	Pressed
1	Left Shift	Pressed
2	Control	Pressed
3	ALT	Pressed
4	Scroll Lock	Active
5	Num Lock	Active
6	Caps Lock	Active
7	Insert	Active

Figure 1. *The keyboard status byte at 0040:0017 indicates the above keyboard states.*

The BIOS keyboard routines are accessed through interrupt 16h. There are about 8 keyboard functions (depending on the model of the PC), and the function number is passed in the AH register. In this case, the function required is number 1 – 'Get Keystroke Status'. This function will return with the zero bit in the flags register set if there is no character ready to be read. In this case, the routine jumps straight to the end and exits. If there is a character, function 0 of interrupt 16h is used to read it, and the routine loops back to the beginning to see if there are any more characters. When the routine terminates there will be no pending character in the keyboard buffer, and the calling program can display the warning message and wait for the operator response.

The second example is a little more complex. This one is a function that will return the status of the keyboard toggles. This information is stored at location 17h of the BIOS data segment, which starts at segment address 40h. The byte at this location is logically ANDed with the supplied mask to provide a non-zero result if the key is pressed or the toggle is active, and a zero result otherwise. Note that this status byte can also be obtained through interrupt 16h – directly reading the BIOS data area is used here simply to provide an example that contrasts with the previous example.

The QuickBasic code for this function here is –

```
DEF SEG = &H40
Result% = PEEK(&H17) AND Mask%
```

Using the library routine the code becomes –

```
Result% = KbdStat% (Mask%)
```

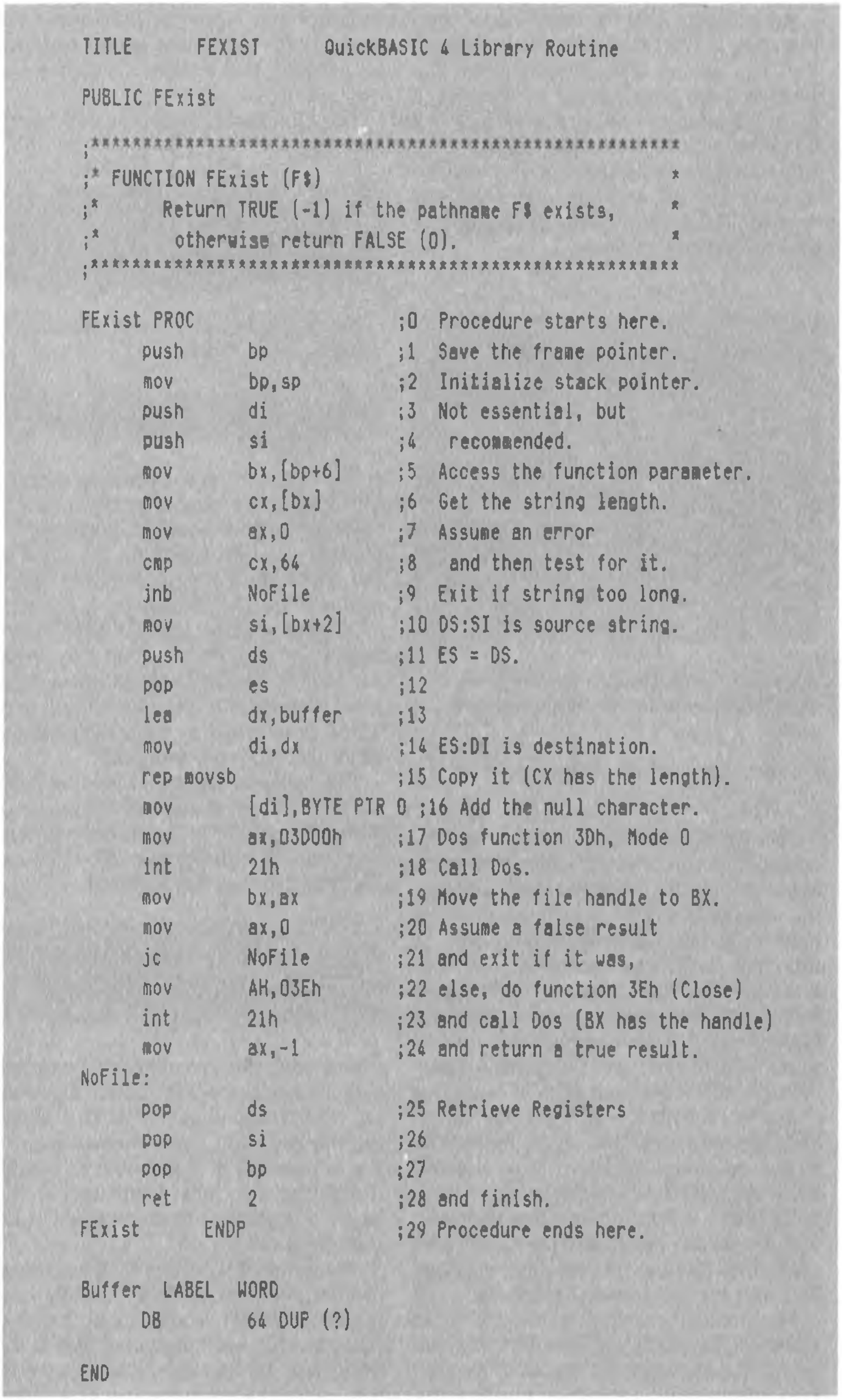
The code for the routine is as simple as the description suggests. Because a parameter will be accessed using BP, this register must be saved on the stack and the contents of the stack pointer copied into it. Then the ES register is set to the BIOS data segment address (40h) and the byte at offset 17h is copied into register AX. This is logically ANDed with the supplied parameter to produce a result in AX, which is returned to the calling program when the routine terminates.

The value of Mask% should be set so that the required keys are tested. For instance, to see if either shift key is currently pressed, use a mask value of 3 – bits 0 and 1 of the byte set. For instance–

```
IF KbdStat%(3) THEN PRINT 'One or
      both shift keys'
```

– note that this technique is particularly useful when cursor keys and function keys are being trapped using ON KEY. By testing the keyboard status immediately the key is trapped, it is possible to separate the shifted, Control and Alt states of these keys – something the QuickBasic ON KEY statement cannot do.

The range of possible BIOS functions is too large to cover here. Usually, BIOS functions will be used as a part of a larger, more complex, routine rather than in their own right. For instance, a windowing system will make extensive use of BIOS routines, as well as direct reads and writes of video memory. It is unlikely that the BIOS disk routines would prove very useful, but the keyboard, video, comms port, printer, and system status BIOS calls can be put to good use providing programs with easy access to these facilities. □



```
TITLE      FEXIST      QuickBASIC 4 Library Routine

PUBLIC FExist

;**********************************************************
;* FUNCTION FExist (F$)                                   *
;*      Return TRUE (-1) if the pathname F$ exists,       *
;*       otherwise return FALSE (0).                      *
;**********************************************************

FExist PROC                    ;0  Procedure starts here.
      push      bp             ;1  Save the frame pointer.
      mov       bp,sp          ;2  Initialize stack pointer.
      push      di             ;3  Not essential, but
      push      si             ;4   recommended.
      mov       bx,[bp+6]      ;5  Access the function parameter.
      mov       cx,[bx]        ;6  Get the string length.
      mov       ax,0           ;7  Assume an error
      cmp       cx,64          ;8   and then test for it.
      jnb       NoFile         ;9  Exit if string too long.
      mov       si,[bx+2]      ;10 DS:SI is source string.
      push      ds             ;11 ES = DS.
      pop       es             ;12
      lea       dx,buffer      ;13
      mov       di,dx          ;14 ES:DI is destination.
      rep movsb                ;15 Copy it (CX has the length).
      mov       [di],BYTE PTR 0 ;16 Add the null character.
      mov       ax,03D00h      ;17 Dos function 3Dh, Mode 0
      int       21h            ;18 Call Dos.
      mov       bx,ax          ;19 Move the file handle to BX.
      mov       ax,0           ;20 Assume a false result
      jc        NoFile         ;21 and exit if it was,
      mov       AH,03Eh        ;22 else, do function 3Eh (Close)
      int       21h            ;23 and call Dos (BX has the handle)
      mov       ax,-1          ;24 and return a true result.
NoFile:
      pop       ds             ;25 Retrieve Registers
      pop       si             ;26
      pop       bp             ;27
      ret       2              ;28 and finish.
FExist      ENDP               ;29 Procedure ends here.

Buffer  LABEL  WORD
      DB        64 DUP (?)

END
```

Listing 2. The library routine is simply the statement CLRKBD. Like the FEXIST% function presented above, the library routine will follow the logic of the QuickBasic version. The BIOS will be interrogated to discover if there is a keystroke ready. If there is, it will be collected and discarded. This will continue until the BIOS says that there are no more keystrokes, and the routine will terminate. The routine takes no arguments and returns no results.

(This 14-part series is concluded in *Learning with Your Computer No 4*).

No Experience Necessary

GEOWORKS PRO

If you've used an A.T.M. lately you've probably got all the experience you'll ever need to use GeoWorks Pro!

GeoWorks Pro delivers two great products for the price of one, GeoWorks Ensemble and Quattro Pro Special Edition. Together, these two multi award-winning products give you the easiest and complete integrated solution for your small business. Word processing, Spell checker, Spreadsheet with charting and database, Desktop publishing, Communications, Address book, Label and Banner maker, Appointment scheduler, File and disk manager, and much, much more.

GEOWORKS

COMPUTERMATE™
(australia) pty. ltd.
ACN 003 868 497

9 High St., Mt. Kuring-Gai, NSW 2080.
Ph: (02) 457 8388 Fax: (02) 457 8739.

A "PANE-less" G.U.I.

for <u>under</u>

TWENTY FIVE BUCKS

GEOWORKS

Instead of wading through long technical descriptions to find out about **GeoWorks Pro,** our *full Working Model* lets you get your hands on it. We think you'll find it so versatile, easy, fast and fun to use, that you'll want to buy the real thing.* The *Working Model* does everything that **GeoWorks Pro** does, except save your files.

You get the entire multi-tasking, windowing, graphical environment PC/GEOS, including:

- **GeoWrite word processor.**
- **GeoDraw drawing program.**
- **GeoManager disk and file manager.**
- **GeoDex address book & label maker.**
- **GeoPlanner schedule planner.**
- **GeoComm communications program.**
- **Six Software "Appliances" designed for the beginner, including Banner Maker.**
- **Solitaire & Tetris Games.**
- **and much, much more.**

With the Working Model you can print out everything you create in dazzling detail on any dot matrix printers!

* The full purchase price of the Working Model will be rebated when you return your Working Model and buy GeoWorks Pro ($249)
GeoWorks Pro includes Quattro Pro SE Spreadsheet

Please send me the 5.25"/3.5" (cross out whichever inapplicable) GeoWorks Pro Working Model.
I enclose $24.95 or charge my Credit Card.

Signed ______________________ PLEASE TICK ✓ □ Bankcard □ Mastercard □ Visa

□□□□ □□□□ □□□□ □□□□ __/__ EXPIRY

NAME: ______________________

ADDRESS: ______________________

______________ POSTCODE: __________

PH (W): __________ PH (H): __________

Mail to:
Computermate/GeoWorks Working Model,
9 High St Mt Kuring-Gai, NSW 2080 Australia
Phone: (02) 457 8388 Fax: (02) 457 8739

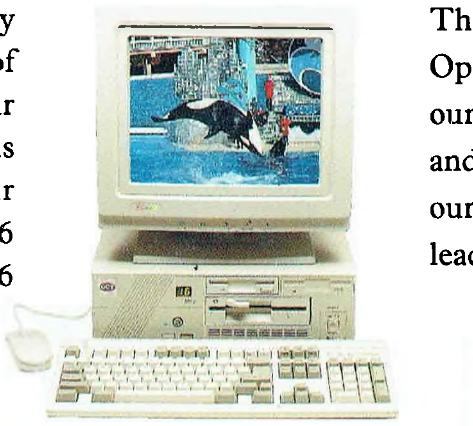
System Requirements: IBM PC/XT/AT or 100% Compatible, 512k RAM, 5.25" or 3.5" Floppy Disk Drive, Mouse and Hard Disk Drive

Trademarks belong to their respective registered owners

Five Year Warranty On Demand!

Optima's new 2 or 5 year warranty option allows you the flexibility to choose between either our standard 2 year or our optional 5 year warranty.

Optima Computer Technology lets you choose the type of warranty that best suits your computer needs. Two years is now standard on all of our computers, from a base 286 workstation through to a 486 50mhz EISA System. A five year warranty, backed by Optima Computer Technology one of Australia's largest PC manufacturers, is available as an option.



OCT OPTIMA
COMPUTER TECHNOLOGY

The stringent tests which all Optima computers undergo at our factory ensure their quality and reliability and also support our growing reputation as a leading Australian supplier of the latest technology. To find your nearest dealer from our Australia-wide network, simply phone either – Sydney **(02) 638 1222** or Brisbane **(07) 262 8377.**